Essentials of Scientific Computing

Kevin Cooper

October 19, 2016

Contents

Chapter 0

Introduction

In the twenty-first century science flows in a river of computing technology. Scientists store their measurements in sophisticated databases that they access and manage across networks. They use powerful software to analyze and model those data. They publish the results in a finished form, both in print and on the World Wide Web. A scientist or mathematician who is ignorant of all the tools of this computing technology will soon drown. Certainly, there are computing support personnel that can throw a life ring, perhaps graduate students can be hired to manage a raft of computers, but in the end, a scientist who does not have at least a rudimentary understanding of the tools of modern scientific computing will not navigate the river successfully.

In this document we discuss some of the most common tools needed by a modern investigator in mathematical and scientific disciplines. The topics are quite varied, ranging from tools used in reporting results to the tools actually used in the analysis and modeling of data. In most cases we present software and techniques that are actually in use now. In a few cases, we try to do some foreshadowing, bringing in tools we think will be used heavily in the relatively near future.

To illustrate this information we will examine the tasks required of a person who wants to analyze data from a database somewhere on the Internet, and then present the results in a mathematical paper or on the World Wide Web. We will necessarily consider a very simple example, so that anyone with a beginning undergraduate background in mathematics can follow it. The stress will not be on the science or mathematics, but instead on the computing tools required to make the final report happen.

The proper order of the topics could be debated. It might be argued that analysis comes first, and writing and reporting comes later. On the other hand, one might need to use the Internet extensively early in order to download data from a remote site, or in monitoring equipment at that site. We will not worry excessively about the order in which the tools would be used in actually producing the science and reports about it. We instead begin by discussing the Internet itself and the ideas that make it work. We then discuss the World Wide

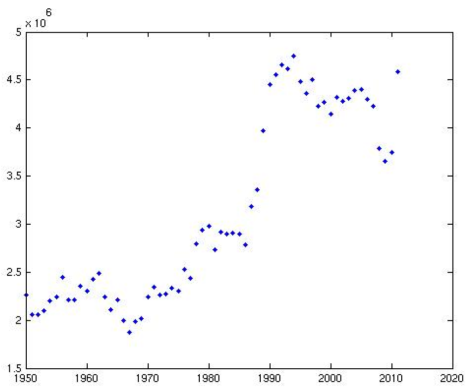


Figure 1: Total landings of commercial fisheries in the U.S. (metric tons)

Web and the tools that make it a powerful instrument for retrieving and disseminating data, and particularly scientific and mathematical information. Following that we will spend some time discussing the defacto standard for typesetting mathematical documents. Many areas of science do not use $\LaTeX$, but since it remains the most sophisticated software for typesetting documents that use mathematical notation, and because its syntax is widely used among other programs such as Microsoft Word and LibreOffice, we will present it in detail.

We finish with a discussion of several commonly-used mathematical/statistical packages: Matlab, Maple, Python/Scipy/Sympy, and R. These represent a wide array of modern scientific software packages. We cannot possibly do justice to all of the software in current use for statistical and mathematical analysis, but these packages and languages use relatively standard syntax and are very widely available.

0.1 Example – the problem

Consider a simple problem that an undergraduate in a scientific discipline might face. Ann needs to download some data from the National Oceanic and Atmospheric Administration's web site. She must then model the data using simple least squares analysis. When she has found some coefficients and created some of the plots, she must post her results on the World Wide Web, so that others in her group can collaborate with her and exchange information. Finally, she and the others in her group must write a report on the project.

Annie's assignment is to visit the National Marine Fisheries Service web site [3] and download data for the total landings of commercial fish for the period from 1950 through 2011. She will then create a web page with those data displayed graphically along with links to the site from which she downloaded the data, and her prospectus for modeling the data. Later in the semester, as she learns more about the mathematics and computation required to model the data, she will do so using a popular mathematical software package, and then display those results graphically at the web site she create earlier. Finally, she will write a paper describing her methods and results to submit as a term project.

The first step of her project requires Ann to know something about computers, the Internet, and other topics pertaining to the World Wide Web. In the second stage, she must learn how to typeset documents using HTML and MathML, and to create graphics using SVG. After that she must know enough about extant software packages to choose a good one in which to do her analysis, and then actually to carry out that analysis. Finally, she must know LaTeX well enough to write her report. We will cover each of these topics in detail, referring in every case to this example problem.

Chapter 1

Networking Concepts

Thirty years ago, a computer network was composed of a single large machine connected by long sets of wires to a number of small green-screen terminals. The terminals were only capable of displaying twenty-three lines of text using a restricted character set. The most exciting form of communication that took place over the computer network was the Unix utility called "talk", which permitted two users of the same computer to type text messages to each other.

Clearly, times have changed. No one can work in science or engineering today without a computer nearby. Electronic mail, typesetting, and web services accelerate the flow of information to speeds unimaginable just a few decades ago. A variety of data compression schemes, coupled with large-bandwidth networks, make transferring images just as easy as transferring text. Pocket telephones today have more power and memory than that mainframe that supported 100 green-screen terminals in 1980.

What is the Internet? The name itself says much about it: it is a network that connects networks. In other words, while one may have a local network anywhere – in the office, in the home – that network is a world unto itself unless it is connected to other networks. The Internet comprises the infrastructure and software that permits you to connect from your local network to machines all over the world. The infrastructure is composed of cables, network devices that switch and route the signals to the next station, and the collection of languages and programs that provide the protocols for these devices to talk to each other.

There are many kinds of media used to create computer networks, from small cables made of copper to fiber optic strands. To some extent, the properties of the media determine the types of packets that are sent across them, but different packet technologies may use similar media. For example, fiber optic cables are used for FDDI packets on some networks, for ATM transmissions on others, and for fast Ethernet packets on others still. The key to the Internet is that all of these different media, transmitting different kinds of packets, may still communicate through the use of standardized protocols, such as TCP/IP, and at higher levels, FTP and SSH.

1.1 Background

Why is it that so many things in computing seem to be organized into fours, eights, and sixteens? What is the difference between Ethernet and Internet? In spite of efforts to make networks and the Internet transparent to users, there remain many places where a little bit of understanding of electronics and binary arithmetic can help you make some sense of what is happening.

At the bottom of the network you use is the physical layer. This is the actual cable, network interface cards, and other hardware that is required to make the network function. There are many different styles of cable used on computer networks, and different kinds of communication used on those cables, but many of them rely on sending low-voltage electrical signals over the cables. Those voltages are always interpreted in a binary sense - either on or off. If a voltage is ”on”, then it is interpreted as a one, while if the voltage is ”off”, then it is interpreted as a zero. Of course, there are network media that use light instead of electrical signals to record those ”on” and ”off” notions, but logically, the idea is the same.

Since every signal is composed fundamentally of ones and zeros, then mathematically we clearly are working in base two. For anyone familiar with number systems, this should present no problem, but let’s review it here. When we write a base two numeral, the only digits permissible are ones and zeros. As in a decimal representation of numbers, the position of the digits has a meaning in terms of the power of the base that the digit is multiplied by. More simply, in this case the rightmost digit multiplies 2^0, the second digit from the right multiplies 2^1, the third from the right multiplies 2^2, and so on. When there is ambiguity it is convenient to write the base of the number being expressed using a subscript. For example we can write $10_2 = 2_{10}$. This cryptic expression indicates that $10_2 = 1 \times 2^1 + 0 \times 2^0 = 2$, with the last number written in base ten. Example 1.1 gives more instances of this.

Example 1.1

$$\begin{aligned}
100_2 &= 1 \times 2^2 + 0 \times 2^1 + 0 \times 2^0 = 4_{10} \\
101_2 &= 1 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 5_{10} \\
111_2 &= 1 \times 2^2 + 1 \times 2^1 + 1 \times 2^0 = 7_{10} \\
1001_2 &= 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 = 9_{10} \\
10011001_2 &= 1 \times 2^7 + 0 \times 2^6 + 0 \times 2^5 + 1 \times 2^4 \\
&\quad + 1 \times 2^3 + 0 \times 2^2 + 0 \times 2^1 + 1 \times 2^0 \\
&= 153_{10}.
\end{aligned}$$

In computing, each digit of the binary representation is called a *bit*. These bits are typically organized further into groups of eight, called either *bytes* when we are discussing the size of files, or in the context of networking, *octets*. We typically do not want to write eight binary digits for every number we discuss, but eight binary digits are sufficient to represent numbers from 0 to 255, inclusive, such numbers appear prominently in all aspects of computation. There are several ways to write these numbers. Obviously we can use a decimal representation as we have above, but it is equally common in computing to use a hexadecimal representation.

Hexadecimal means:“base sixteen”. Writing hexadecimal numbers is requires more digits – we need single digits to represent ten through fifteen. It is traditional to use the letters A through F for this purpose. In other words, if we were to count in hexadecimal, it would go like this: 0, 1, 2, 3, 4, 5, 6, 7, 8, 9, A, B, C, D, E, F, 10, 11,. . . This all fits together nicely with base two: $16 = 2^4$. Thus, any hexadecimal digit may be represented as a four digit binary number (four bits), and conversely, four bits may be represented by a single hexadecimal digit. Moreover, $256 = 16^2$, so that we may use one hexadecimal number to represent the upper four bits of an octet, and a second hexadecimal number to represent the lower four bits. Thus, any number from 0 to 255 inclusive can be represented by a two-digit hexadecimal number. The position of the digit again signifies the the power of sixteen by which it is to be multiplied. Example 1.2 illustrates this further.

Example 1.2

$$\begin{aligned}10000000_2 &= 128_{10} = 80_{16};\\ 10011001_2 &= 153_{10} = 99_{16};\\ 11110001_2 &= 241_{10} = F1_{16};\\ 11010_2 &= 26_{10} = 1A_{16}.\end{aligned}$$

It is easy to convert between binary and hex. Given any binary number, we only have to group its digits into fours, and then write the hex representation for each of those groups. For instance, in Example 1.2 the third line shows that $11110001_2 = F1_{16}$. Observe that the first four digits of 11110001_2 are $1111 = F_{16}$. The second four digits are $0001 = 1_{16}$. Grouping the digits into fours allows us to write down the hex representation immediately. Again, in the last line of Example 1.2 we see that $11010_2 = 1A_{16}$. This time the binary number is not divided evenly by fours, so we pad it on the left using zeroes: we write $00011010_2 = 11010_2$. Written that way, we break it into fours: $0001_2 = 1_{16}$ and $1010_2 = A_{16}$.

Converting from hex to binary is just as easy. Each hex digit can be written as four binary digits, and place holding is preserved. For example, suppose we want to write the binary representation for $B3_{16}$. We know that $B_{16} = 1011_2$, and $3_{16} = 0011_2$, so that $B3_{16} = 10110011_2$. Again, $3B_{16} = 0011\,1011_2 = 111011_2$.

We have seen that it is not very hard to convert from hex to decimal (base ten): we just expand the expression gigit by digit as we described above. For example, we can write $B3_{16} = B \times 16^1 + 3 \times 16^0 = 11 \times 16 + 3 \times 1 = 179_{10}$. In the same way $3B_{16} = 3 \times 16 + 11 \times 1 = 59_{10}$. By contrast, converting from decimal to binary or hex is not all that easy. To convert a number from decimal to hex, we first have to know how many hex digits will be required. Observe that 179_{10} has a three-digit decimal representation, but only a two-digit hex representation, so we see that there is an issue here.

Since each hex digit of a representation is a coefficient of 16 to some power, it follows that finding out how many hex digits are required to represent a given decimal number amounts to finding the largest power of 16 that is less than or equal to that number. For our example we see that $16^0 = 1 \leq 179, 16^1 = 16 \leq 179$, but $16^2 = 256 > 179$. Thus the hex representation of 179 will have 16^0 and 16^1 terms, but no 16^2 term, so we need only two hex digits.

Next, we have to figure out what the coefficients of those terms are. We start with the highest degree terms: for our current example 16^1. How many 16^1s are there in 179? The answer is 11, because $11 \times 16^1 = 176 \leq 179$. Thus the first digit in the hex representation is eleven – B in hex. To the the rest of the digits, we subtract the part we have accounted for so far from the original number: $179 - 11 \times 16^1 = 3$. Thus, our hex representation for 179 is $B3_{16}$. Another illustration of this can be seen in Example 1.3.

Example 1.3 Convert 312_{10} to hex and binary.

We note that $16^0 = 1 \leq 312; 16^1 = 16 \leq 312; 16^2 = 256 \leq 312;$, but $16^3 = 4096 > 312$, so our hex conversion will have coefficients for the $16^0, 16^1$, and 16^2 terms: three digits.

How many 16^2s are there in 312? There is just one, so the first digit is 1. That accounts for 256 of the 312; we still need digits to account for $312 - 256 = 56$.

How many 16^1s are there in 56? There are three: $3 \times 16 = 48$. Now there are only $56 - 48 = 8$ unaccounted for. The last digit is 8.

We conclude that $312_{10} = 138_{16}$. To convert to binary, just write each hex digit as four bits: $138_{16} = 0001\,0011\,1000_2 = 100111000_2$.

Hexadecimal numbers – hex, for short – appear often in computing as a way to shorten the notation and ease the computations for binary numbers. For example, hexadecimal numbers are typically used in writing addresses for Ethernet networks, as we see in the next section.

1.1.1 Ethernet

The most common type of local network is called Ethernet. This word actually describes a collection of different media and signal types. All these different technologies are united in the sense that they are organized logically around the idea of *packets*. A packet is an ordered collection of ones and zeros that have a particular significance – if you will, an electrical or optical signal with a beginning, some content, and an end.

Each packet contains an address for the local machine that is its immediate destination. That address is called the Ethernet address, or more often, the MAC (Media Access Control) address. The MAC address is a 48-bit binary number. In writing the number, we don't want to use the space and time required to put down each of the ones and zeros, so it is organized into six octets. Typically, each octet is written using hexadecimal notation, and the octets are separated by colons. Thus, MAC addresses are typically written in forms such as 08:00:A0:B1:18:03 or 00:05:15:64:68:F0. We see that the 48-bit address is written as twelve hexadecimal digits. As we noted above, each hex digit represents four bits of information.

It is worth noting that MAC addresses are purchased in blocks of 16,777,216 by the manufacturers of Ethernet interfaces. This means that the first three octets in the MAC address identify the manufacturer of an Ethernet card, while the last three octets are simply a number from within that sequence. Thus, each MAC address uniquely identifies an Ethernet interface ("network card").

Every Ethernet packet contains either a MAC address of its immediate destination, or an address that indicates that it is intended to be received by every computer on the local network. In this way, the receiving computer understands which of the packets on the network are intended for it, and which are not. Each Ethernet packet also contains the address of the computer that sent it, so that if a reply is needed, it may be sent.

There are many other types of computer networks, but we do not need to consider them or the details of Ethernet networks. Our point here was simply to show the way that the physical network of cables and appliances fits together with the logical network of ones and zeros, expressed using hex notation.

1.1.2 Colors

Another place where hexadecimal numbers show up often is in the specification of colors on the Internet. Most likely you have seen a monitor that claims to display millions of colors, or perhaps claims exactly 16,777,216 colors. It is not a coincidence that this number is exactly $256^3 = 2^8 \times 2^8 \times 2^8$. Colors are specified on computer screens as combinations of red, green, and blue, or in computer-speak: RGB. The intensity of each color component is typically an eight-bit number - an octet. Again, this is frequently written using hexadecimal notation. Thus, it is common to see a color written as a six-digit hexadecimal number. The first two digits denote the intensity of the red component of the color, the second two denote the green component, and the third pair denote the intensity of the blue. Some examples of color specifications follow.

Example 1.4

FFFFFF – maximum red, maximum green, maximum blue – white;
000000 – no red, no green, no blue – black;
FF0000 – maximum red, no green, no blue – red;
600000 – less than half red, no green, no blue – dark red;
006000 – no red, less than half green, no blue – dark green;
FFFF00 – maximum red, maximum green, no blue – yellow;
E0E0E0 – equal amounts red, green, blue – light gray
E0E0FF – E0E0E0 + 00001F, i.e. light gray + a little bit of blue – light blue.

The last item from the example points out a peculiarity of color specifications: the smallest number from among the red, green, and blue components determines the amount of white (or gray, if you will) in the color. In particular, the last color should be viewed as E0E0E0+00001F, or in words, as light gray with a little bit of blue.

1.2 Packets and the TCP/IP Suite

In July of 1961, a mathematician at the Massachusetts Institute of Technology wrote a thesis discussing the idea of “packet switching” [6], though it was not called that at the time. This was a landmark, inasmuch as this idea forms the basis for most computer networks since then. There are two basic forms of communication between computers. Connection-oriented networks work by forming fixed connections between the computers, dedicated to the conversation between those computers. During the time that the connection exists, the path between the computers is unchangeable and guaranteed. No other computer can interfere with the interaction. On the other hand, it is expensive to maintain enough lines to carry the connections required, and once the lines available are all in use, then no further connections may be established. Worse yet, if any aspect of the connection between the two devices is damaged, whether the cable or any small electrical component, then the connection is severed. The telephone network of the 20th century is an example of a connection-oriented network. To see the chief disadvantage of connection-oriented networks, just think of the busy signals that used to be common on telephone networks.

In contrast, a packet-switched network is connectionless. The basic idea behind packet switching is to send information in small batches, called (of course) packets. A message may be broken into these small pieces and wrapped in information about its source and destination, and its place in the sequence of packets to be sent. Each packet might travel by separate routes to its destination, where information it contains may be used to reassemble it with other packets to form the entire message.

Each packet can be formed in such a way as to show whether it remained error-free, and indeed can be put together to allow a certain level of error correction. If a packet was damaged or lost, it can be sent anew, without having to send the entire message.

The advantage of this approach is clear enough: a single wire may be shared among many computers. The disadvantage is equally clear: if too many computers try to use the line, then all communication will be slower – indeed, all traffic will be slowed. To imagine this, one might think of a party. If there are not too many people at the party, then small groups may form to carry on conversations, each party ignoring what it hears from the other. However, if the party is larger, then the conversationalists have increasing difficulty hearing each other, and frequently speakers must repeat themselves. The actual network has a worse time with this, inasmuch as the capacity for carrying the packets has an absolute limit (unlike the party).

These concepts were developed through the decade of the 1960s in three separate laboratories, independently. Indeed, the researchers involved were unaware of each others' work until 1967. In 1967, a paper was presented providing a view of what a packet-switched network might look like, and the ARPANET2 was born. ARPA, or the Advanced Research Projects Agency, was the U.S. governmental body responsible for a number of advances in computer science in the 1960s and 1970s. The ARPANET was its creation, perhaps its most notable achievement. By the end of 1969, four computers were able to communicate with one another on this precursor of the Internet.

If a message is to be broken into packets, obviously there must be some uniform way of doing this so that the receiving machine can reassemble the message from the packets. The initial ARPANET used rules encapsulated in NCP – Network Control Protocol – to perform the tasks of disassembly and reassembly. However it became clear early that this protocol was limited, and that it would be necessary to extend it or to create a new protocol governing this aspect of the network. For this reason, in 1972 Bob Kahn and others formulated principles for a new set of packet-handling protocols, known now as Transmission Control Protocol and Internet Protocol - TCP/IP [10]. This combination is now the basis for much of the traffic on the Internet.

While what Kahn and Cerf formulated was a single system called Transmission Control Program, it was quickly made more modular, so that there are several separate protocols involved here. IP (Internet Protocol) is the part that deals with ”long range” addressing, while TCP (Transmission Control Protocol) manages the connectionless session. In other words, IP is used to provide each packet with the information needed to get it to its final destination, while TCP is at a higher level, breaking the message into packets, keeping track of whether all the packets have been delivered, requesting that packets be sent anew, if necessary, and reassembling them at the destination.

We do not need to know the details of the TCP protocol here, so we only provide a very brief summary of the ideas. Simply put, TCP breaks the data into pieces, encapsulates those pieces into packets, together with information regarding their position within the whole document and the IP address of their source and destination, and then sends off the packets. Along the way, some of the packets are lost and some arrive out of sequence. Nonetheless, TCP on the receiving computer sends an acknowledgment packet for every packet received. Some of those packets are lost in turn, but as those acknowledgments that do arrive come in, TCP on the sending computer checks off the packets it knows arrived. After a timeout period, it sends those for which it has not received acknowledgments again. Since some of the original packets did arrive at the destination, and it was the acknowledgments that went awry, the result is that some packets are received twice or more. TCP on the receiving machine simply discards the excess packets, sends acknowledgments again, and begins to reassemble the content.

Thus, the job of the sending computer is to keep sending packets until all are acknowledged, while the receiving computer must send acknowledgments for each packet that comes in. This simple-sounding combination of tasks is actually quite complicated to maintain in practice, and thus TCP is not a trivial protocol by any means. This also means that the network is filled with packets that do not contain content of the information being transferred. Finally, it is a good deal of work for both machines to keep track of all the incoming and outgoing packets.

On the other hand, not every network application requires the reliable session that TCP provides. For such applications, other less reliable protocols suffice. The User Datagram Protocol (UDP) is used for many file services and similar applications in which it is not necessary to have a "session". For example, in Unix, the network file service uses UDP to maintain contact with a server. In that situation, if files are not actually being transferred then all that is required is for the client machine to keep track of the server machine, to be sure that it remains up and is willing to serve the file system in question. UDP does not use acknowledgment packets to maintain the session, and it does not keep track of the order of incoming packets. The advantage is speed. However, UDP still uses IP to manage the addressing, and so it is considered to be part of the TCP/IP suite of protocols.

In order to illustrate these ideas, consider an email message from a computer called mymachine.edu destined for yourmachine.com. The email program assembles the message and passes it, together with the address of the destination, down to the TCP protocol. TCP breaks the message into packets, and places the address of the destination computer into each packet. It also places information into each packet regarding where that packet fits into the whole. After that, it puts each packet onto the network. Each packet moves along the network to various nodes, called routers, that determine where it should next be sent. As a result, the packets may take different routes to yourmachine.com. For example, the first packet may go to yourmachine.com via a router in St. Louis, while the second packet sent may go through Kansas City. The third packet may go via a route that is currently experiencing errors, and become corrupted. The result of this is that the second packet sent is the first to arrive at yourmachine.com, followed closely by the first one sent. Each time a packet arrives, the TCP software on yourmachine .com sends an acknowledgment packet back to mymachine.edu. When mymachine.edu has waited some period of time without receiving the acknowledgment of packet number three, it sends that packet again. Eventually TCP on yourmachine.com recognizes that all of the packets have arrived, assembles them back into the entire message, and passes that to its email handling program for delivery.

There are a couple of external things required for all of this to work. There must be a coherent addressing system for the Internet;
there must be a computer somewhere that understands how to translate names of computers into those addresses. Finally, each of the computers at either end of the transaction must have software to handle the TCP protocol, called a TCP stack.

The addresses used are called (surprise!) IP addresses. IP addresses are very simple - they comprise four octets. In another way of looking at it, an IP address is an ordered collection of four numbers between zero and 255, inclusive. For reasons that are mired in the history of the Internet, when IP addresses are given it is traditional to represent each octet of the address in decimal form, instead of its hexadecimal form. Thus, typical IP addresses look like 191.16.1.1, or 134.121.43.233.

Humans generally have more trouble remembering sets of numbers such as those in IP addresses than they do with names. For that reason, we have all grown used to the idea of giving the names for the computers we want to contact: for example, www.amazon.com, or www.math.wsu.edu. Those names do not go into the packets sent to those computers. Instead, the program sending the packets first asks a nameserver to find the IP address associated with the name given. The nameserver is simply a program on a computer that is in charge of the names of certain groups of machines. No single computer knows the names of all the computers on the Internet; they simply know which other computers to contact in order to get those names and addresses.

In recent years it has become clear that there are far more devices on the Internet than there are traditional IP addresses. In other words, there are at most $2^{32} = 4,294,967,296$ four-octet IP addressses, while there are more devices than that on line today. This problem is partially dealt with using segmentation in networks, including private addresses behind firewalls, but in the long run, we just need more addresses. This need has led to the introduction of a new IP address system. The older one is now called IPV4, while the new protocol is called IPV6. IPV6 address comprise eight groups of four-hex-digit numbers. Thus, there are 2^{128} IPV6 addresses. It should be enough to last a while.

Ethernet or other packets with IP information alone would have all they need to get to their destinations. Unfortunately, networks are a dangerous place: variable electric fields or poor connections can introduce errors in packets; two or more packets can try to share the same network segment, leading to a collision; routers and other network devices can malfunction, causing packets to be lost. If packets only carried addressing information, only a fraction of them would get to their ultimate destinations. The TCP protocol was developed to provide reliable transport of collections of packets over an unreliable network.

Reference 1.1

Internet Terms

Bit – A single binary digit.

DNS – Domain Name Service. This is the system that governs requests and answers associating Internet names with IP addresses.

Ethernet – A specific type of network arrangement used by most universities and businesses within their facilities. It is packet-switched, and can be sent over a variety of media, at various speeds that depend on the media.

IP address – A four-octet number that identifies a unique computing system on the Internet. IP stands for Internet Protocol.

Nameserver – A computer that runs a program that knows how to associate an IP address with an Internet name. Nameservers are sometimes known as DNS servers.

Octet – Packets are generally formed from groups of eight bits - ones or zeros. These groups of eight are typically called bytes in the computer world, but in computer networks, for reasons that are unclear, they are usually called octets instead.

Protocol – A language that computer programs use to communicate with each other. The word refers specifically to languages used over a network.

RFC – Request For Comments. These form the alarmingly democratic structure governing standards and documentation for the Internet.

Port – Some network transport protocols associate certain programs with numbers. Even though the network hardware receives all of the packets through the same connection, it sorts the packets into logical mail slots for their destination programs according to these numbers. These logical mail slots are called ports.

1.3 Public Key Encryption

Public key encryption actually relies on two keys: one is public, while the other is private. You are free to send the public key to anyone and they can use its to encrypt data to send back to you. On the other hand, once encrypted the data can only be unencrypted using your private key. Thus, in the case of SSH, you send your public key to any machine with which you want to communicate, and thereafter the session is encrypted using that public key. Only your computer can understand the communication after it is encrypted, because only your computer knows the private key.

The ideas behind this are rooted in number-theoretic ideas. For example, observe that $(2^3)^5 \bmod 7 = 1 = (2^5)^3 \bmod 7$, and in general $(g^a)^b \bmod p = (g^b)^a \bmod p$, when p is prime. We can use simple relations of this nature to create complicated systems of public key encryption.

The simple idea is that two computers A and B agree on values for g and p. Computer A then chooses a secret key a, while B chooses its own secret key b. A sends $g^a \bmod p$ to B, while B sends $g^b \bmod p$ to A. The result is that:

- The world can know g and p.
- A cracker can snoop $g^a \bmod p$ and $g^b \bmod p$ by intercepting the communications between computer A and computer B, but without knowing a and b, it doesn't help him.
- A and B both know $g^{ab} \bmod p$. This is a secret key.
- Only A knows a; Only B knows b.

That secret key can then be used in a variety of ways to encode, and later decode a message between the two computers. In this way a private message can be sent across a public network while remaining understood only by the sender and receiver.

1.4 Running commands on a remote computer

TCP/IP provides the means of getting your information from one computer to another, but what is that information? TCP/IP carries the information from many different, higher-level programs. The first of these to be standardized was the *telnet* protocol. Telnet very simply allows you to execute commands on a remote computer. This means more and less than a generation raised on the Microsoft Windows interface might understand. When telnet first came into use, all computers interacted with a user purely through a prompt at which the user would type text commands. For example, to list her files, the user might type the Unix command `ls -l`. The telnet protocol did no more than codify a means of allowing the user to execute such commands on a machine far away.

To use telnet, you do not need to do any more than start a command line on your local computer, and then type `telnet remotecomputer`, where `remotecomputer` is the Internet name or IP address of the computer you want to log into. If the computer accepts the connection, then it should invite you to enter your user name and password, and it should give you a command prompt. You are free to run any commands on that remote computer that you would be able to if you were sitting at its console.

We should note here that there are actually two things happening in a telnet session. First, you are running a program on your local computer which in turn its talking to a program on the remote computer. The program on your computer is called a Telnet client. In general, when we use the word client we mean a program on your computer. The program on the remote computer that respondents to your Telnet request could be called a Telnet server, but it is usually referred to using Unix parlance, and is called a telnet daemon. The two programs communicate using the Telnet protocol. It is the Telnet protocol which is standardized through the RFC process. Thus, while there are many different telnet clients which may incorporate more or fewer features, there is effectively only one Telnet protocol. All of those Telnet clients should use the same protocol.

Some computers are not good at starting shells where you can type "telnet". Instead, such computers frequently allow you to start a telnet client in a new window. In this case, you click a menu item or double-click an icon labeled "Telnet" or "Terminal" or something related. In particular, Windows used to ship with a default telnet client that ran in its own window on the screen. You could start it in any of several ways, but it is probably easiest to click [Start→Run] and type telnet remotecomputer in the text box of the dialog that results.

Sometimes you want to log into the remote computer using an ID different from the one on your own computer. Telnet provides for this by including a switch to allow you to change IDs. The syntax is `telnet remotecomputer -l userid`, where user ID is the name of the user that you want to use of the remote computer. In other words, this command will log you into the computer remotecomputer using the name userid.

There are three points worth making here. First, it is probably important to distinguish between the telnet client and the telnet protocol. The client is an actual program that runs on your local machine that knows how to use the telnet protocol to communicate with remote computers. The protocol is the high- level language that the client uses. The protocol is standardized in RFCs, but there are as many different clients as there are dandelions in your yard. The second point is that the ability to run a telnet client does not correlate with the ability to run a telnet server.

This leads to the third point. Telnet grew out of the Unix world, and developed very early in the Internet world. That is why it is command-line-oriented, but it also explains the comparative lack of security it embodies. If your computer provides telnet services, that means that it has a TCP port open to the world. That telnet port may be used by anyone to try to log into your computer. The only security you have lies in requiring users to give a password to log in to your computer. Unfortunately, even that security is minimal. Anyone who has access to any computer on your local network can run a program called a

Software 1.2

There are a number of SSH clients for Microsoft Windows operating systems, but the one that has become a defacto standard is called PuTTY. It is free software, found at
`http://www.chiark.greenend.org.uk/~sgtatham/putty/`.
Linux and Mac users can just launch SSH from a command line: make a terminal and type `ssh accountname@host.machine`.

"packet sniffer" that examines all packets that go by. When it finds the one (or more) containing your password, it logs it. The owner of the sniffer comes back later to examine the log (or has it transmitted to him), and then he has your password. This is only the most obvious way to compromise a telnet service. There are many others. To address this glaring insecurity in telnet, Secure Shell protocols have been developed. These are called collectively SSH, and the only thing they add to the telnet service is encryption. If you use SSH, then instead of sending your password and other information across the network in plain text, you send an encrypted version. Presumably the person running the sniffer cannot decrypt the password in a short enough time for it to be useful, so you are more secure.

Using SSH is almost completely analogous to using telnet. In particular, the basic command to use SSH is `ssh remotemachine`, where `remotemachine` is replaced by the name or IP address of the computer you mean to contact, e.g. www.wsu.edu. Again, if you want to log in using a user ID that is different from the one on your own machine,then you may use the -l option, or you may use the command `ssh othername@remotemachine`.

SSH works using public-key encryption, as discussed in Section 1.3. There are two versions of SSH. SSH 1 uses 56-bit encryption. That means that the public key is a number that can be expressed using 56 binary digits. SSH 1 has become vulnerable to certain attacks. In particular, it has long been vulnerable to a so-called *man-in-the-middle* attack. In this, a program masquerades as an SSH 1 client and server, reads your public keys, and then operates an SSH session on your behalf, using its own private keys. To you it looks like an ordinary SSH session, but in truth everything you send or receive can be decrypted by the man-in-the-middle. SSH 2 is a newer protocol that uses 128 bit encryption. It is currently less vulnerable to these attacks, though you must always make a choice when you first contact a machine as to whether to accept its public key, and at that time a man-in-the-middle is possible regardless of encryption. When you need to distinguish between SSH 1 and SSH 2 it is probably best to call it as `ssh -2 remotemachine`.

Software 1.3

FTP clients are very common, and generally show more sophistication than SSH clients. In particular, most modern FTP clients use a graphical windowed interface, allowing a user to select file names on one machine and use a mouse to drag those to an area representing the other machine. Following are a couple of common FTP/SFTP clients. They are both free software.

Filezilla – *http://filezilla-project.org*
This is a multi-platform client that supports FTP and SFTP.

WinSCP – *http://winscp.net*
A Windows-only client with some nice features.

1.5 File transfer

Telnet and SSH allow you to run commands on a remote machine. Unfortunately, that doesn't permit you to run the commands and then send results back to your home computer for analysis. This is a kind of task that occurs very often in scientific applications; you want to use some powerful machine at another university, but you want to produce the graphics and reports on your desktop machine or on your home computer. To facilitate the transfer of files from one computer into another, file transfer protocol (FTP) was created.

File transfer protocol actually allows you to run many commands on the remote computer. For example, you can list the contents of a directory, you can change directories, and sometimes you can even move files from place to place on that computer. On the other hand, it does not give you complete control on the remote machine. It is designed to allow you to manipulate files and to transfer them to your computer but not to allow you to run programs or perform other tasks on the remote computer.

FTP first appeared an in 1971 in RFC 114. That means that FTP antedates TCP. Indeed, the first versions of FTP used data transfer protocol to maintain the connections required to send files across networks. Version 2 of FTP appeared in 1971 also, followed by version 3 in 1972. The version of FTP we use today is numbered 5.1, and the standards for that date from 1985.

FTP is similar to telnet in that there is an FTP client running on your computer, and an FTP daemon running on the remote computer. The FTP daemon communicates with your computer using the FTP protocol. In short, we must again be careful to distinguish between the protocol and client programs.

There are many sophisticated client programs that use the FTP protocol. However, at bottom they all use a relatively small command set that is shared with more primitive FTP clients. They all do the same jobs. They all have roughly the same capabilities.

We will proceed by discussing the use of a primitive FTP client, and later finish this section by discussing one example of a more sophisticated client. We will not discuss details of the protocol itself.

Before we discuss the command set, we must observe that FTP itself is rarely used anymore. Like telnet, the problem with FTP is that it is not encrypted. Again, it is easy for someone on the network to see every packet go by, and thereby steal information. For that reason the SFTP protocol was created. It is analogous to SSH, in that it encrypts an entire FTP session. The encryption is more or less transparent to the user – SFTP clients have interfaces identical to older FTP clients. You should only use SFTP, and henceforth we will assume that you do.

1.5.1 Basic FTP commands

Because FTP is so old, many SFTP clients still use a command-line interface. Such interfaces are very primitive, but are still effective today. In this section, we discuss some of the basic commands found in any FTP or SFTP client. In this discussion we will suppose that we are looking for some free software that plots graphs of functions. We will look for the software on the computer freesoftware.org. That computer is imaginary. Then again, it probably isn't. Since many of the earliest FTP clients ran on Unix computers, the command-line interface to those clients is very like that of Unix. In particular, commands have the form commandname option1 option2... The options are separated by one or more spaces.

Obviously, the first thing we have to do with SFTP is to make a connection to a remote computer. We initiate this process in any of several ways. For example, for almost a command-line interface we type `sftp freesoftware.org`. This establishes an SFTP connection with the machine freesoftware.org, which asks us for authentication information. We enter our user ID and password to establish the connection. Alternatively, we could simply type `sftp` without any mention of the name of the remote computer. At that point we get an SFTP prompt, at which we enter the command `o freesoftware.org`. The command `o` stands for open, meaning that we wish to open a connection to remote machine. Once again the remote computer will prompt you for your user name and password to complete the connection.

We should note that there are many so-called anonymous FTP sites on the Internet. They are called anonymous because you do not need to have an account on those computers in order to make FTP connections. Indeed, anyone can log into those computers and retrieve files from them. Such sites are usually used to distribute free software or documentation about software. Usually they invite you to enter "anonymous" as login ID, and to use your email address as your password, so that they have some record of who is using the site. We will suppose that freesoftware.org is such a computer, so that after we opened the connection to that computer, we typed "anonymous" for our user ID, and yourname@yourmachine.edu for the password.

Now that we are connected to the computer, we need to see what directories are available from which we can retrieve files. We can always list the contents of a directory by using the `ls` command. The command `ls` simply stands for "list". Of course, SFTP is designed to transfer files, so that in typing `ls`, we are actually asking the SFTP server to create a file composed of the contents of the current directory and send it to us. On the other hand we are asking the local SFTP client to take that file and display it on the screen. The file contains the result of the Unix command `/bin/ls`. We see that that the pub directory looks promising, so we can change to that directory by using the `cd` command. "cd" stands for change directory. We type `cd pub`. After that we use the `ls` command again to list the contents of that directory, and continue in that way until we find what we want. When we find the directory that contains the file we want, then we can retrieve that file using the `get` command. The `get` command brings the file to our computer and puts it on the current directory in our local computer. You might ask, "what is the current directory?" Since we're assuming that we are running the FTP client program from a command line, then the current directory is simply the directory we were in when we started ftp. Thus, supposing that the file we want is called libraries.doc, then to retrieve the file from the directory at freesoftware.org that we are looking at, and to put it in the current directory on the local computer, we need only type `get gnuplot.tgz`.

It frequently happens that we start the SFTP client without thinking about what directory we are in. We find the file we want to retrieve, but then realize that it will go to the wrong directory, or even to a directory that we are not allowed to write to. We need to change the local directory to deal with this problem. To do that, we use the `lcd` command. "lcd" stands for "local change directory". In our example, we had been looking for plotting programs on our Unix machine, in the /usr/local/bin directory. We found that we lacked any decent programs for our task, so we logged into the anonymous FTP server to get that library. Unfortunately, now our local current directory is /usr/local/bin, on which we do not have permission to write. To change it, we just type the command `lcd /home/mydirectory` to get to our own directory, and then type `get gnuplot`.

The `get` command can take two arguments, if you want to specify the local file on which to place what gets transferred. For example, to get the gnuplot file above without changing directories all the time, you might use the command `get /usr/local/bin/gnuplot /home/mydirectory/gnuplot.mine`. This gets the file from the remote directory, and puts it directly on your home directory with the name "gnuplot.mine".

Unfortunately, we might find after we get the file that we cannot use it. The reason is that the default method of transfer for primitive FTP clients is to send all files as if they were composed of plain text. We don't need to know the details of what that means, aside from noting that when the file is compressed, or is a program, or is anything other than plain text, then the transfer is incomprehensible to the receiving computer. To deal with this, we must tell this client that the file to be transferred is not a text file. We do that by using the command `binary`. In other words, before the `get` command, we type `binary`. Most recent versions of SFTP clients use binary mode by default – it is rare to need to enter this command anymore.

Sometimes we want to get several files off of the remote computer. If they have similar names, we can use the `mget` command together with wildcard characters to get the collection. The name `mget` stands for “multiple get”. In our current example, we realize when we go back to freesoftware.com that in addition to using plain text mode to transfer the file we wanted, we also neglected to get a collection of interface functions that gnuplot needs. Therefore, after typing binary, we get the files using the command `mget gnuplot*`. This command brings us both the gnuplot file and the gnuplot_x11 file. The asterisk in the above command is called a wildcard. It signifies that gnuplot* matches any string that starts with the word “gnuplot”. Thus it matches the names gnuplot, gnuplot_x11, gnuplot_my_foot, and gnuplot_is_a.very_fine_plotting.program. It does not matter how many characters we put in place of the asterisk – it matches anything.

When we are finished transferring files, we exit the SFTP client using the command `quit`. This simply terminates the connection. It does not save anything, or ask you for verification.

Reference 1.4

FTP commands

binary – this command tells FTP to interpret the file in a raw form. The default is to treat the file as plain text. We use this command when we want to send executable files or compressed files. It takes no arguments.

cd – To change directories on the remote machine, use this command. To change directories relative to the current directory on the remote computer, use the form e.g. cd mydirectory. To make a change to an absolute path, preface the directory specification by a slash: e.g. `cd /subdirectory`.

get – This is the basic command to retrieve a file from the remote computer. The syntax is get remote_file local_file. You may leave off the second file name, if you want to store the file on the current local directory using the same name.

hash – This peculiar little command causes hash marks to be printed at intervals during the transfer of a file. The marks give you an idea of how fast the transfer is taking place, and give you assurance that the operation is proceeding, or let you know when it has stagnated.

lcd – To change directories on the local machine, use the lcd (local change directory) command. Again, use either relative or absolute specifications for the path to the directory.

lls – This command lists the contents of the current directory on the local computer.

ls – This command lists the contents of the current directory on the remote computer.

mget – You can transfer multiple files from the remote machine to your local computer using this command. Typically, one would use this in conjunction with wildcard characters. For example, `mget myfile*` would get all files whose names start with the letters "myfile".

mput – Similarly to mget, the mput command allows you to transfer several files from your local computer to the remote machine.

put – This is the basic command to put a single file onto the remote site. The syntax is to `put local_file remote_file`.

Chapter 2

The World Wide Web

When Tim Berners-Lee developed the first browser program in the early 1990s, it changed the Internet forever. Before the browser, our experience of the Internet consisted entirely of interaction with primitive command-line tools. We had to memorize a number of arcane and limited commands in order to interact with various separate programs that performed different functions. The browser brought a number of different Internet protocols into a single program, and made our interaction with that program much simpler. The result was that command shells, file transfer, and text and graphics display could all take place in one program, giving a life and ease of use to the network that had not previously been there.

2.1 The Browser

What is this thing that made a computer network come alive? It is nothing but a program that is capable of fetching and displaying a large variety of data. There have always been many different kinds of files available on computers. The problem for users was that it was difficult to determine which files were programs and which were input to programs, or output from programs, or images to be displayed. Moreover, different file formats required different methods of transfer from one computer to another. Data files might be used on many different operating systems, while computer programs were highly dependent on operating systems and hardware architecture. In order to view formatted text or an image file, one needed to install the viewing program on one's own computer.

The problems in networked computing were exacerbated by the plethora of protocols required to transfer data from one computer to another. Some of these protocols are listed in Table 2.1. The different protocols had different functions, and the programs that implemented them used different syntax. This made the problems of networked computing comparable to those experienced by a world traveler. For example, suppose that you are a researcher in physics at UCLA in 1988. You want to run some programs on a computer at Argonne National Laboratories, and then analyze the data on computers at your university. You must first log on to the local computer, which runs an operating system called VMS, which you must use fluently. Then, you must start a program that supports the telnet protocol to log in to the machine at Argonne. Next, you use the command line for the operating system at the Argonne machine (Unix) to run your program, producing a data file. Now you must transfer the file back to UCLA, so you log out of the telnet application, and start one that supports FTP. You use a different syntax in this program to retrieve the file you created at Argonne, and you are finally ready to analyze the data using a plotting package on your machine. To summarize, to accomplish this transaction, you must know the syntax of VMS, Unix, a telnet application, an FTP application, and your local data analysis/graphics program.

Table 2.1: Common Protocols for World Wide Web Communication

Protocol	Description
FTP	File transfer protocol - used to transfer files of all types from one computer to another.
HTTP	Hypertext Transfer Protocol - used to transfer files from one machine to another.
HTTPS	Hypertext Transfer Protocol Secure - used to transfer files from one machine to another over an encrypted connection.
SFTP	SSH File transfer protocol - used to transfer files of all types from one computer to another over an encrypted connection.
SMTP	Simple Mail Transfer Protocol - the language of electronic mail.

Browser programs did not change these facts, but they were able to listen over the network and speak the language of the machines they found. They could understand when to format output, when to display an image, and when to save a binary file. This ability to deal with multiple protocols, and to interpret many different file formats opened up the internet to such vastly expanded possibilities that it took on a new name: the World-Wide Web. In modern times, of course,“surfing the Web” is an activity pursued by everyone, from schoolchildren to participants in elder hostels, from American students to Malaysian businessmen. This access is the direct result of the invention of the browser and the attendant server software and infrastructure.

2.1.1 Browser Motivation

There are really two parts to the act of surfing the Web: it requires two computers running programs that can speak to one another. The first is the web server. The server is a computer or program that listens constantly for requests for files, and sends those files to the requesting program. The part that you interact with is, of course, the *browser*. The browser is, in a sense, the boss of the transaction. The browser makes the request, and the server is but an aide who fetches and sends the information. The browser has millions of such secretaries at its disposal. On the other hand, the computer where the browser runs typically does not need to be as big or sophisticated as the server computer, since the server must store large quantities of information and listen for and handle a wide variety of requests from many different browsers.

You have probably already noticed that we use the word server in two different ways in this discussion. On the one hand, the server is an entire computer, complete with an operating system that is willing to offer information through various TCP ports that are associated with various protocols. On the other hand, we frequently use the word server to mean specifically the program that listens for and responds to hypertext transfer protocol. Thus, it is possible for your browser to contact the server computer without ever talking to the HTTP server. In order to attempt to avoid confusion, we will try always to refer to the server computer. When we use the term “server” without qualification, we mean the HTTP server program.

2.1.2 URLs

The heart of the transaction between the server computer and browser is the Uniform Resource Locator (URL). This is the information that the browser uses to find the required file on the appropriate directory on the server, and to transfer that information via the appropriate protocol. The URL has four basic parts: a scheme specification; a machine identifier, possibly combined with a TCP port specification; a path component to indicate the location of the file on the server machine; and finally, either bookmarks within the file or input to the program that runs on the server. The format of these fields looks like this:

```
scheme://server.identification:port/path/to/file?query_string#bookmark
```

You have probably typed URLs into your browser manually before. It is easy to do using a text box on the toolbar of most browsers.

The scheme tells the browser, in essence, what means to use in contacting the server computer. This also carries implications for the TCP port that is used. Typically, the server computer runs a number of different programs, one of which will handle FTP requests, with perhaps another for gopher requests, another for SSH, and another for HTTP. The browser program is able to negotiate with several of those different server programs to get the information it needs. When the scheme refers to a network communication like these, the name of the network protocol to be used occupies the scheme slot.

The most common protocol used by browsers is HTTP: Hypertext Transfer Protocol. This program allows files to be transferred, but also negotiates the format in which those files are transferred, and provides information about what to do with the files after they are delivered. Unlike FTP, which delivers a single file, HTTP can send several files of different types: plain text, formatted text, images, and so on. Note also that when you type the URL into the browser directly, you do not usually need to type `http` if that is the protocol you want to use - it is usually the default protocol for the browser.

Another common protocol you will have seen is HTTPS, denoting secure HTTP. This is simply the HTTP protocol encrypted using Secure Sockets Layer (SSL), a technology that uses the public key encryption discussed in Section 1.3. Other protocols that can sometimes be seen in the scheme field of a URL include FTP, and “mailto”, an interface to the Simple Mail Transfer Protocol (SMTP).

You will also see the word “file” appear in the scheme specification frequently. This does not actually denote a protocol, but instead indicates that the browser should open the file on the local computer directly, without using the network at all. Note that in most browsers you can enter a URL of “file:///” to get a basic file browser on your own computer.

After the protocol specification you must identify the machine you want to contact (unless you used file request instead of a protocol). You may identify the computer using either its Internet name or using its IP address. Typically the machines you want are named www.sitename.com, or something similar. Most often, the www is not the real name of the machine – instead, the machine might have various aliases. To make matters more confusing, in modern times there are many “web hosting services”; companies that sell an address associated with a block of storage on one of their machines. As a result, such a computer might have many thousands of aliases that do not even share a site name. Indeed, the aliases may be associated with individual file systems on the computers. In other words, while we have called this part of the URL the machine specification, it is actually more complicated than that. On the other hand, we do not have to worry about that - we may as well treat this as the name of the computer.
Often enough, it is.

The next part of the URL does not always appear. By default, HTTP is associated with TCP port 80 on the server computer. However, it is possible to set up a server program to listen to almost any port you desire by inserting a colon at the end of the machine name and then the port number you want to use. Ports near 8000 are commonly used. If the port has a number other than 80, you must specify that number after a colon following the computer name. For HTTPS connections, the default port is 443.

Following all of the computer information comes the part of the URL where we tell the location of the file we are trying to reach on the computer. We do this by listing the file name, its directory, and all of the parent directories of that directory, starting from the top-level directory. There are a number of things to note about this. First, the top-level directory is probably not the root directory for the computer. The administrator of the server computer may designate any directory she likes as the “document root”. Only subdirectories of this directory can be viewed using a browser. This provides the first small level of security for the server computer: not all of its files are visible to those using browsers.

Subdirectories of the document root are separated by forward slashes (/), regardless of the syntax in which that is done on the server or browser computers. Thus, on a Microsoft Windows machine, it might be that the file is actually located at `C:\Front Page Web\Web\Mathematics\books.html`, but the URL directory specification would show only `/Mathematics/books.html`.

For each web server program the administrator gives a file name that is to be used as the default page for each directory. Typically, the name chosen is index.html or default.html. Whenever a browser requests a URL that ends with the name of a directory, the server program looks in that directory to see whether it contains a file with the default name. If so, the server delivers that file to the browser.

Anything that appears after the file name is either a target within the file, or input to the program in the file. Targets are specified using a pound sign (#), while input to programs usually follows a question mark. We shall not discuss this further in this section.

Consider the URL `http://math.wsu.edu/Calendar`. This tells the browser to use hypertext transfer protocol to contact a machine named math.wsu.edu, and to look in a subdirectory of the document root called Calendar for a file named index.html (the default). The actual path to the file on that Unix server is `/home/www/math/Calendar/index.html`. It might be that the same machine runs a second server program that listens to port 8080. A URL for a file on in that document tree might be `http://math.wsu.edu:8080/special/contents.html`. Again, the same machine might run an anonymous ftp server listening to port 22. A URL for a document available by that means might have a URL of the form `ftp://math.wsu.edu/ftpdocs/program.exe`.

Most people do not pay much attention to URLs as they surf the Web. However, we have all needed at some point to go to a specific Web site. We may do that in almost any browser by typing the URL into the text box on a toolbar near the top of the browser window, and then pressing the <Enter>key.

2.1.3 Local Customization

By now you are aware that the entire idea behind using a browser is to do as much as possible locally. That idea continues into the presentation of the pages the browser retrieves. The browser allows you to control many aspects of the appearance of pages displayed in the browser, even to the point of overriding some of the aspects of the presentation put in by the page author. In particular, most browsers allow you to specify which fonts you want to use, and colors you prefer for the pages you view. Do this with care, since you can break the pages you view this way. These days web designers have great control over the appearance of their pages, and changing the font size or other aspects of the page can completely ruin the appearance of the pages.

The Cache

The most time-consuming thing a browser does is retrieve pages over a network. Formatting and displaying those pages proceeds very quickly on the local computer, but waiting for those pages to arrive across the network occupies much of the time of the browser, and the person using it. To reduce the time required for that, virtually all browsers use file caches. The idea is simple: the browser stores every page it retrieves locally. Later, if the user wants to return to that page, or if an image from one page is used repeatedly on others, the browser can pull the file or image from its local cache, instead of having to get it over the network again.

This can sometimes cause problems for a browser user. Perhaps you want the original page from the server because it has changed. Perhaps your cache has grown so large that it is interfering with your computer's operations. In the first case, most browsers have a means of forcing the browser to reload from the server. For example, Firefox permits you to hold down the <`Shift`> key while you click the reload button to force the browser to retrieve the page again. In the second case, you might need occasionally to clean out your cache. Doing so saves storage. You may also prevent the cache from becoming too large by controlling its permissible size. For example, in Firefox you can control the size through [Edit→Preferences→Advanced→Cache]

MIME

So many different forms of documents may be transferred over the Internet that browsers (and e-mail tools) need some means of keeping track of them. In particular, some documents need to be opened using a program not included in the browser, some files contain images in different formats, and some files should just be saved to a hard disk. Most browsers use two tools to sort the files out: the filename extension and the MIME type. These are related.

The file name extension is probably the more intuitive of the methods. Typically, file names on Unix and Microsoft operating systems have the form `name.extension`. The name is whatever you want it to be, but the the extension contains information concerning the nature of the contents of the file. For example, the browser would understand that a file named bob.gif contains a digital image in graphics interchange format, and should be displayed directly, while a file named bob.ps contains a document in the Adobe postscript language, and should be passed to an external viewer such as ghostview. The extensions are usually kept short – indeed, in 16-bit Microsoft operating systems the extension was required to be at most three characters. On Unix/Linux machines and 32-bit or 64-bit Microsoft operating systems the extension may have any length, but it is very rare to find an extension longer than four characters.

On the other hand, filenames can lie. You may put your postscript document or GIF image into files with any names you want. For example, if your browser receives a file with the name bob.postscript, it probably will not know how to handle it. It was partially for this reason that MIME (Multipurpose Internet Mail Extensions) was created. MIME provides more direct information about the content of files passed over the Internet, independent of the names of those files. Most Web and email servers are trained to send MIME information, and Web authors may do so explicitly in their documents as well. MIME types consist of two words separated by a slash ("/"). For example, the GIF image we transferred earlier would have a MIME type of "image/gif". The postscript document would report a MIME type of "text/postscript". The browser, in turn, either knows by default or may be trained to recognize these MIME types and to know how to handle them.

There is a small danger in this. Sometimes web programmers specify the MIME types in odd ways. For example, one browser supports MIME types of both `application/ps` and `application/postscript` for Postscript documents. These might be associated with different applications, or with no application at all. This can be frustrating. Sooner or later, you probably need to become familiar with your browser's application preferences configuration.

Interaction and Animation

In the beginning of the World Wide Web it was envisioned that documents would be transferred together with easy ways to retrieve related information. However, it was not very long at all until it became clear that browsers needed to run programs. Web site managers wanted to receive information from people viewing their site, or they wanted to provide means for viewers to interact with their displays. Sometimes they just wanted to make their pages look more lively. Regardless of their motivation, any kind of animation or interaction requires that a program run somewhere, and that program needed to be triggered by the browser of the viewer.

Many ways of allowing browsers to run programs were developed,
which may be grouped into "server-side" programs and "client-side" programs. As you might imagine, server-side programs run on the server machine, at the command of the browser. The most common examples of such programs are Common Gateway Interface (CGI) programs and Java servlets. These are used primarily when the author of the page being viewed wants to receive and record input from the viewer of her pages.

Client-side programs run either directly in the browser, or are interpreted by some program outside of the browser. For example, Java programs run in most popular browsers directly, while Maple worksheets require the program Waterloo Maple to be started by the browser, whereupon the Maple program runs the input file that was retrieved over the Internet.

Running programs is a critical part of scientific use of the Web, but it is also fraught with danger. Any time you allow someone to run a program on your computer,
you take a chance that he will do something you don't want to your machine. Various different client-side programs attempt to protect you in a variety of ways, which we shall detail later. Suffice it, for the moment, to say that you

do not have to allow programs to run on your computer. All browsers must allow you the option to prevent Java, Javascript, and other such programs from running automatically, and many cautious users do this. Firefox even has an addon called *noscript* to prevent any page from running javascript programs in the browser until given permission. Server side programs do not present the same threat to the web browser, but they in turn can be attacked.

This topic is large and important in scientific use of the Internet, and we shall discuss it at some length in later chapters.

2.1.4 Editing Web Pages Using a Browser

Some modern browser programs have HTML editors built into them. Typically, one loads a page in the browser, and then can open that same page in a wysiwyg editor. The editor shows some approximation of what the page will look like when displayed on a browser, and provides toolbars and menus to insert the HTML items you need. We'll create a page identical to the one shown in Section 2.1 to illustrate the process.

HTML editors are very similar, and indeed, look very much like word processor editors of any kind. Typically, there is a menu bar at the top of the editor window featuring the usual suspects for menu items: File, Edit, View, Insert, Format, and so on. Below that we find a toolbar that holds buttons to provide shortcuts to frequently used functions. Below that lies another toolbar containing font specification boxes and paragraph formatting shortcut buttons. Finally, below that we find a window where the text we type appears. This description applies to most of the popular editors. Since these are so similar, and since Mozilla Composer is free and platform-independent, we use that editor to develop our example page. The best way to get the Mozilla Composer is to load SeaMonkey onto your computer. You can get it from `http://www.seamonkey-project.org` [**?**].

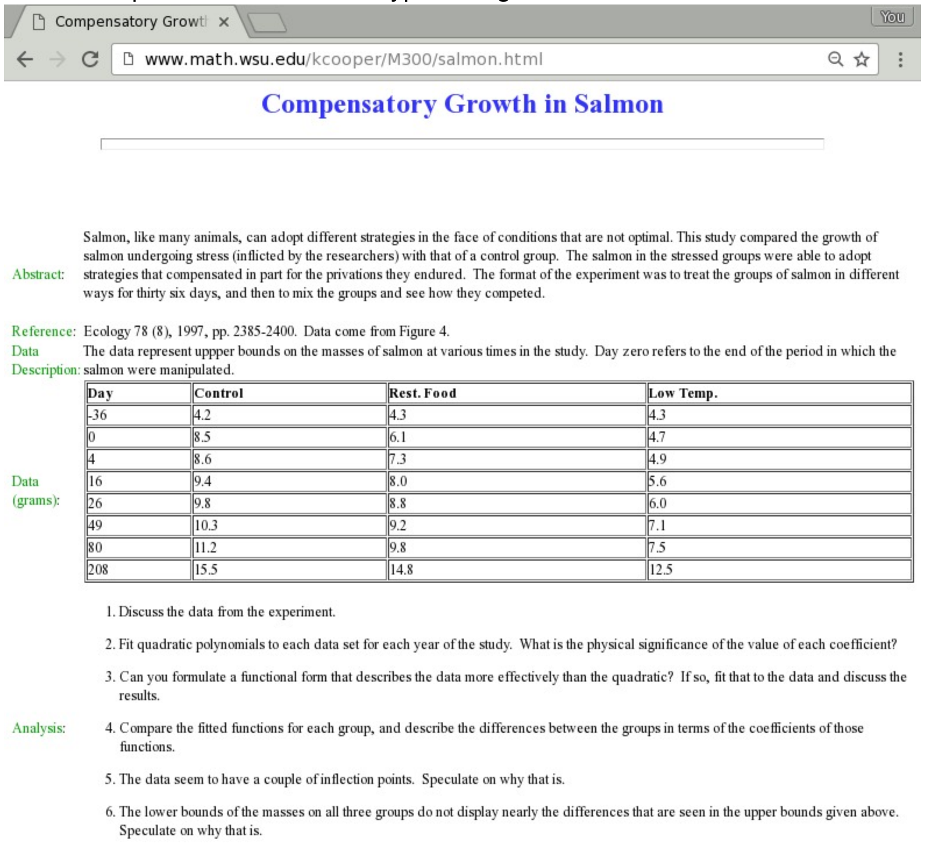
First, we open a blank page. To do that we start Seamonkey, and then click [File→New→Composer Page]. We should see a new window that satisfies the description given above. All we have to do is fill it up now. We could start with the last lines of the page, but we won't. The title is ”Compensatory Growth in Salmon”. We type that into the screen. Obviously, there is no formatting for that text yet.

We could do the formatting simply by picking a font family and size. However, that approach is fraught with problems. What if we are creating the page on a Windows machine and our viewer is using a Unix machine? Our viewer doesn't have an Arial font, and so she sees a typewriter font instead. What if our viewer is not using SeaMonkey, but instead is using some unknown browser program that handles font tags differently, or not at all? We want to be sure that this title looks like a title on all browsers that load this page. What if we specify a small font, but our user has made all of his fonts small to make the most of a small screen? In that case, our fonts might be illegible.

Fortunately, HTML comes with tags that are supposed to represent head-

---

Example 2.1 Below is a part of a Web page that gives an exercise intended for undergraduate science majors. The exercise itself concerns a study from the literature of compensation for adverse conditions in salmon growth. The page itself has a few interesting features.
First, the title appears in an unusual font and color. The horizontal rule underneath it is again different from the default. Finally, all of the text beneath the horizontal rule is set as a table, with item names on the left in a green font, and data and descriptions on the right. You may view the entire page at http://www.math.wsu.edu/kcooper/M300/salmon.html. We will use this page as an example for all of our Web typesetting.



Compensatory Growth in Salmon

Abstract: Salmon, like many animals, can adopt different strategies in the face of conditions that are not optimal. This study compared the growth of salmon undergoing stress (inflicted by the researchers) with that of a control group. The salmon in the stressed groups were able to adopt strategies that compensated in part for the privations they endured. The format of the experiment was to treat the groups of salmon in different ways for thirty six days, and then to mix the groups and see how they competed.

Reference: Ecology 78 (8), 1997, pp. 2385-2400. Data come from Figure 4.

Data Description: The data represent uppper bounds on the masses of salmon at various times in the study. Day zero refers to the end of the period in which the salmon were manipulated.

Data (grams):

Day	Control	Rest. Food	Low Temp.
-36	4.2	4.3	4.3
0	8.5	6.1	4.7
4	8.6	7.3	4.9
16	9.4	8.0	5.6
26	9.8	8.8	6.0
49	10.3	9.2	7.1
80	11.2	9.8	7.5
208	15.5	14.8	12.5

Analysis:

1. Discuss the data from the experiment.
2. Fit quadratic polynomials to each data set for each year of the study. What is the physical significance of the value of each coefficient?
3. Can you formulate a functional form that describes the data more effectively than the quadratic? If so, fit that to the data and discuss the results.
4. Compare the fitted functions for each group, and describe the differences between the groups in terms of the coefficients of those functions.
5. The data seem to have a couple of inflection points. Speculate on why that is.
6. The lower bounds of the masses on all three groups do not display nearly the differences that are seen in the upper bounds given above. Speculate on why that is.

Reference 2.1

Nomenclature

Throughout this text we use certain common terms in a technical sense. The terms, and any actions they describe, are listed below.

Select – whenever we want to apply some formatting to a block of text we must first mark that text in some way. The way we do that is by using the mouse to click the uppermost left edge of the text block, and then holding the mouse button down while we drag it to the bottom right edge of the text block. As we do this, the text should change color to white, in the background should change color to black, highlighting the text we have selected. After that we can apply whatever format specification we wish to the highlighted text.

Keystrokes – we indicate keystrokes by enclosing them in angle brackets. Thus, if we want you to push the "Enter" key, then we write "press <`Enter`>".

Menu – most computer programs now make use of menus. The menu is usually a bar near the top of the window for our application. It typically contains items labeled "File", "Edit", and so on. When we refer to menu items we will use the notation [Item→Inner Item→Inner Inner Item] to indicate the progression of menu items we should click. For example, in most applications we can save a file by clicking [File→Save]. This notation indicates that we use the mouse to click the File menu, and then we click the Save item within that menu.

Pixel – a Picture Element. This is a unit of measurement for objects on a computer display - basically the size of one dot on your screen.

Wysiwyg – an acronym for "What you see is what you get". Typically in typesetting text of any kind, there are several ways to look at the document: some show the typesetting commands, and others show only the finished product. When wysiwyg editors appeared in the decade of the 1980's they were revolutionary - typists could see the product before they printed it.

ers, and which are used uniformly by all browsers. We will render this title text using the largest header format. The header formats may be selected in the list box at the top left of the window. Usually, when Composer starts, this contains the word ”Body Text”. Make sure that the cursor is somewhere in the text you just typed, and then click the little arrow at the right of that list box to see a list of all the format tags from which you may choose. Then click ”Heading 1”. The type for the title should become much larger. If not, check again to be sure that you had the cursor placed somewhere in ”Compensatory Growth...”.

We may change the color of the title text by selecting all of it, then selecting [Format→Text Color]. We chose blue, but you may choose your favorite from among those colors.

The title is still justified left. We want it in the center. To do that, again click in the center of the text and then click [Format→Align→Center] from the menu, or click the button with centered lines from the Format toolbar. Your title should now be centered on the page.

Now we have a title in a large font in our favorite color. Next we insert a horizontal line. Move to the line after the title, and click [Insert→Horizontal Line] from the menu. The line that appears should span the entire width of the page, and is very thin. We prefer a line that only spans 80% of the page, is centered, and is thicker. To change the nature of this line, right-click on the line. A menu appears inviting us to change any of several things: click “Horizontal Line Properties”. A dialog box should appear allowing us to control the appearance of the line. Enter 10 in the box labeled “Height”, and enter 80 in the box labeled width. The width setting should already be given as a percentage of the window, but note that we could change the specification to pixels in the box to the right of the width specification. The height specification is not a percentage of the screen height, but instead is measured in pixels. Note also that there is a radio button where we could change the alignment of the line to the left or right. It should be in the center by default. When you are finished, click the OK button.

Sometimes the editor does not cooperate well. For example, it might be difficult to right-click the horizontal line precisely when it is thin, or it might be that a menu is disabled. In that case, you can always change properties in the HTML directly. Click the HTML Source tab at the bottom of the window, and look for the ¡hr¿ tag. In that you can change the width to 80%, and the height to 10px. When all else fails, the HTML Source tab is a good fallback for changing things.

Back on the Normal tab, put in a couple of blank lines by pressing the <`Enter`> key a couple of times. We should note that this does more than you might think. The browser actually ignores blank lines, so in order to get one, you must put certain HTML tags into your document. Pressing <`Enter`> inserts a line break, ¡br¿ in HTML. You can look at that on the HTML Source tab.

Now we are ready for the content of the document, which is entirely composed of a table. It may not look to you like a table, but observe that all of the labels on the left are aligned, and the text and smaller table on the right are likewise aligned. This may only be done with a table. Click [Insert→Table...] to produce a dialog box where you may enter various characteristics of the table you want. First, specify two columns. You may put in the number of rows you want, but it is not important - we can create new rows very easily. Let the width of the table stay at 100% of the window. Below that there is a text box that controls the table border. Set the Border to 0 pixels. When you are finished, click the "OK" button. You should see a placeholder table outlined in red on your page. The red will not be visible on the actual page - you can go to the preview tab at the bottom of the Composer window to be sure.

We are now ready to put the content in our table. Click in the upper-leftmost cell of the table, and type "Abstract:". Observe that the left column grows wider as you type. That is because the default behavior of the table is to adjust the width of columns for the amount of content in them. To move to the next cell to the right, just use the <`Tab`> key. Type some of the abstract as given in the example above: "Salmon, like many animals...". When you are tired of typing, you may move to the next cell using <`Tab`>. Perhaps you made a table with only one row, because you believed us when we said it was easy to make more rows. In that case, there is no next cell. That's no problem: just hit <`Tab`> anyway, and Composer makes a new row for you. Your cursor will be in the leftmost cell of that row, where you may type "Reference:".

In the figure from Section 2.3, we notice that the entries in the left column of the table are in green font. It would be nice if we could change the entire column to use a green font, but we can't. This is one of a number of issues that arise from the very simplicity of HTML. Instead, we need laboriously to select each label in the left-hand column, and then click [Format→Text Color] to obtain a palette from which we may choose our favorite color for the font. When we learn about Cascading Style Sheets later in this chapter, we should be able to do this formatting more simply.

At some point, you will need to make a change to the format of the table. You may do that any time by clicking with the right mouse button inside the table. A menu appears in which you may click "Table Properties". You will notice that the Table dialog box actually has three tabs (see them at the top of the page) labeled "Cell", "Row", and "Column". You may control the text alignment, background color, and other properties using these three tags. It is a nice touch that some changes can be made by column. Tables in HTML are organized by row, and there is no HTML tag to allow control of entire columns together. You might consider this to be one of the many weaknesses of HTML - and the creators of some editors apparently agree.

We should note here also that you may insert rows and columns using some of these same features. When you right-click in the table, a menu appears. This is the same menu that allowed us to click "Table Cell Properties" before. To add rows or columns, click "Table Insert" and then click the menu item that corresponds to the thing you want to insert, and where.

The next feature of note in the page is the table that appears in the row labeled "Data". That is, in fact, the second table on the page, and it lies inside the main table. This is not a problem: we put our cursor in the rightmost cell of the ”Data” row, and again click [Table Insert→Table...]. This time when the table dialog box comes up, make four columns, and leave the border at width 1.

Both of the table dialog boxes - the one you get from [Insert→Table→Table...] and the one you get on right-clicking the table - contain a button to trigger advanced options. The border specification pertains to the outside visible borders of the table. If you set it to zero, that is the same as forcing the table not to have a border. If you set the value to e.g. 4 or 8, then the outside border of the table will be thick, but the inside borders between cells should not be affected. Those spaces between the lines dividing cells are controlled by the cell spacing. The cell padding specification applies to the white space between the content of a cell and the cell boundary. In other words, if you set the cell padding to 10 pixels, then there will be a lot of space around your text within a cell, while if you have no padding, then your text should push right up to the edge of the table cell. Try putting in various different values for these settings, and note the differences.

Type the header row for the data table, and then one row of data. You can make a new row in the table at any time by clicking in the lower rightmost cell of the table and hitting your <`Tab`> key. Note that the header row is supposed to look different than the data rows. You can change some characteristics of the entire header row using the table dialog box. In particular, you can select the row and set the cell style to “Header”. This should cause the headers to appear in a bold face.

The last feature of the page is a list of questions at the bottom. The questions are in a numbered list. The numbers in the list are the result of the action of the browser program - not the HTML programmer. Make a cell in your larger table to put the list in, and then click the button on the Format toolbar that shows numbers with horizontal lines to their right. The first number of the list should appear. Type the first question. When you are finished, press <`Enter`>, and a new pound number will appear. You may repeat this process as needed.

If you started typing the question before you clicked the number to do the enumeration, have no fear. Just select the paragraphs you want to be numbered, and then click the number button on the toolbar. Composer will interpret each new paragraph as a line to be numbered. If you should make a mistake and number a line you don’t want to appear as part of the list, then click in that line and then click the number button again. The numbering will be removed.

To save your work on the local computer, click [File→Save]. The menu item [File→Publish] saves your work on a remote computer. To use the latter, you will need to specify the name of the remote computer, and the details of your account there. See the discussion of ftp in the previous chapter.

To reiterate, this discussion used Mozilla Composer for its instructions, but the kinds of commands found here are almost universal. The button labels may change, the details of the dialog boxes may be different, but for the most part, the specifications will be similar.

Reference 2.2

Mozilla Composer Commands

Here is a brief synopsis of some of the most common tasks in Mozilla Composer, and a very brief discussion of how to perform them. It is not intended to be exhaustive.

Boldface – select the text you wish to be in boldface, then click the button on the Format toolbar that shows the letter A in a bold face. Alternatively, you may also select the text and then click [Format?Style?Bold].

Colors – To change the background or text colors on your page, use the menu item [Format?Page Colors and Properties]. This brings up a dialog box in which you may set the colors. For example, to change the background color for your page, first click the radio button labeled ”Use custom colors”. Next, click the button below it labeled ”Background...”. Select the color you like best from the palette provided. You can use analogous procedures to change the colors for text, links, and so on.

Heading – Click the list box at the left of the second row of tool bars - ordinarily this has the entry ”Normal” in it. You should see a listing of styles to use. Choose a heading style to your liking. ”Heading 1” is a larger font, ”Heading 5” is a smaller font. If you have not already typed the heading, then typing now will produce text in that heading style. If you already typed the text you want to be in the heading, then select it before you use the Heading list box.

Horizontal rule – click the button on the toolbar indicating a horizontal rule. You may also click insert horizontal line from the menu.

Images – to insert an image click the button on the first toolbar that shows a small rectangle with three colorful objects inside it. This brings up a dialog box in which you are required to enter several pieces of information. The first is the name of the file containing image. If you not created such a file that you cannot put the image in your page. If you have a file then you may either type its name in the box provided at the top of the dialog box, or click the button labeled ”Browse...”, after which you may click to the proper directory and file that you want. It may be that this is all is required, however if you want special formatting for the image, or other alternatives, you must enter them here. A discussion of some of the options appears later in the text.

Italics – select the text you wish to be in italics, then click the button on the second toolbar that shows the letter A in a slanted face. You may also use the menu [Format→Style→Italic].

Itemized list – click the button on the second toolbar showing three so-called bullets followed by three horizontal lines. After this, every time you press <`Enter`> you’ll get a new bullet.

Reference 2.3

Mozilla Composer Commands

Links – Of course, the real power of the Internet is in allowing links to pages and other sites. If you want to use one in your pages, first type the label that will represent the link on your page. Then select that label text, and click on the button with the picture of a chain link on the first toolbar. This opens a dialog box in which you must enter link information. If the file is local to the machine on which you are working, then you can use the button labeled "browse file" to find a way to the file and enter it. If it is not local, you must type in the actual URL, e.g. http://www.mysite.com/fun.html. When you're finished, click the OK button.

Numbered list – click the button on the second toolbar showing the numbers 1 2 and 3 followed by horizontal lines. After that, every time you press <`Enter`> you'll get the next number in the sequence from which you are working, where you can type another line.

New Page – Click [File→New→Blank Page].

Save – Click [File→Save]. If it is the first time you have saved the file, this brings up a dialog box in which you must enter the file name. The file name should always end in ".html" or ".htm".

Table – To insert a table at the cursor, click the button on the first toolbar with an image of a table on it (two rows, three columns). This brings up a dialog box where you may specify attributes of the table. The only critical thing to specify here is the number of columns - you can always add rows later.

Text (no special formatting) – Just type.

Underline – select the text you wish to appear with an underline, and then click the button on the second toolbar that shows the letter A underlined. You might choose instead to use the menu item [Format→Style→Underline].

2.2 Markup Languages

A markup language is a set of instructions that can be embedded in text to be typeset that tell how to do that typesetting. Virtually all modern typesetting methods take this approach. It is possible to imagine a typesetting structure that would have one file composed entirely of text to be typeset, and another file with instructions of the nature of "indent the first line each paragraph, separate paragraphs by a 10pt space, italicize the fourth word on line 1", and so on, but that would be quite inefficient, and probably difficult to maintain.

By contrast a markup language typically has shorthand commands to do something like this.

```
Ignore end-of-line characters:
Indent paragraphs:
Set left and right margins to 1.5 inch:
Start typesetting:
Fishing is an activity enjoyed by many people.
Typically it requires a
italicize this: rod
and
italicize this: reel,
which are used to cast line tipped with a
specialized hook into the water.
The hook is typically camouflaged with some
sort of lure that fish presumably find attractive;
either live bait, or an artificial creation.
start a new paragraph:
When a fish bites the lure, the hook catches its mouth,
at which point the fisherman,
make this bold face: if awake and sober,
reels in line until he can catch the
fish using his hands or a net.
End typesetting:
```

Obviously the computer needs a more specific set of instructions than those given above, instructions that are easier to distinguish from the text to be typeset. Nonetheless, we see quickly how easy it should be to do sophisticated typesetting, once we understand a little about the commands a given markup language uses. The commands in a markup language are often called "tags".

There are a number of standard features of most markup languages. They usually make some assumptions about routine chores in typesetting. For example, most markup languages would automatically ignore end-of-line characters as we required above, and would choose default values for margins, default fonts, word and letter spacings, and so on. Virtually all markup languages divide their files into two sections: a preamble giving instructions for typesetting the text; and the body of the document, which still includes instructions for typesetting certain smaller features of the document. And virtually all markup languages use special characters to alert the computer when an instruction is coming, instead of text to be typeset.

We will at least mention several markup languages here, including HTML, XHTML, XML, SGML, MathML, SVG, and $\LaTeX$. This list is not exhaustive.

2.3 Hypertext Markup Language

From the inception of the World Wide Web, it was clear that there was a need for a language for the documents to be transferred that would be common to all browser programs. The language needed to satisfy several important criteria.

1. It needed to be text-based; no files with non-printing characters would do.

2. It needed to be simple, since there were no sophisticated editors with point-and-click interfaces to help the authors of the time.

3. It needed to allow for variations in the way browsers chose to present the styles and sizes it specified - in particular, it could not be too choosy about fonts.

There were models for such languages already in existence at that time: they were called markup languages. A markup language is basically any language that satisfies the criteria above: it is purely based on text, and mixes human-readable formatting instructions with the content text. The two prominent examples of markup languages of that time were called TeX and SGML. TeX was a language developed at Stanford University by a group headed by Donald Knuth. It and its variants constituted a defacto standard for mathematical typesetting by the late 1980s, and indeed, remains the standard today. Unfortunately, TeX is not particularly simple, nor does it allow for the kind of variation that would be required for screen presentation of documents on different kinds of browsers on different operating systems. TeX is quite picky about fonts. SGML suffered from many of the same problems.

These criteria and the need for something simpler than the extant markup languages led early researchers to develop HyperText Markup Language, known by its acronym: HTML. HTML was largely derived from SGML, but had a severely limited command set for simplicity. This made files more compact and documents easier to write. It later also led to a number of regrettable proprietary developments of markup text; but that is another story.

We have already seen that there are many ways of creating HTML code now without knowing any of the actual language, but there is much on the Web that does not make much sense unless one knows HTML. Moreover, it is difficult and inadvisable to write pages for the Web using only a proprietary HTML editor, due to the tendency of those editors to be too dependent on specific platforms or hard-coded styles. We'll look at HTML here both for its own sake, and as a simple first example of a markup language.

2.3.1 HTML Principles

The basic rules for HTML are simple, however they are tempered by a concern for consistency with more modern web typesetting languages. In particular, HTML has an XML-compatible counterpart, called XHTML. XHTML has a simple but stricter syntax, and in this century, HTML coding has adopted those standards, even though XHTML was never fully adopted by the Web development community, and has been superseded by the HTML5 standard.

- Commands in HTML are framed in “angle brackets”. The angle brackets are, in fact, the “less than” and “greater than” symbols on the keyboard. Thus each command in the text appears as e.g. `<command>`. Commands in HTML are frequently called “tags”. The tag is the entire structure comprising the angle brackets and command name.
- Each HTML tag initiates a change in formatting that remains in effect until the tag is closed. Thus, each HTML directive really has three parts: the tag that initiates the directive, the content to which the formatting applies, and the tag that closes the formatting directive. Closing tags look almost like the opening tags, differing only in having a slash “/” between the left angle bracket and the command name. For example, a paragraph begins with the tag `<p>` and ends with the tag `</p>`.
- Sometimes it is permissible to leave off the ending tag, since the browser can figure out what is meant by the context. For example, if you type `<p>` to begin a paragraph, and then use the `<p>` again to begin the following paragraph, then the browser can interpret those tags as denoting two separate paragraphs, because it knows that paragraph tags cannot nest. While this is permissible, it is considered to be bad style. It is not permissible in XHTML.
- Certain tags do not enclose any text. For example, the break tag `<br>` simply makes a new line. In XHTML these tags may be entered either as e.g. `<br></br>`, or for brevity as `<br/>`. We pedantically adopt the latter form in this document. In ordinary HTML and HTML5, no slashes are required.
- Tags must be nested properly. If one wanted to italicize a heading, one might open a group as `<h2><i>`. In that case, you must end the grouping with `<i><h2>`, i.e. the last format specified must be ended first. Again, most browsers contain code to correct errors for HTML, but in XHTML, failing to nest tags properly would prevent display of a page.
- You may use either upper- or lower-case letters for HTML tags, but XHTML enforces the use of lower-case only, so it has become standard to use lower-case for HTML tags.

It is standard in all markup languages to divide documents into two major parts. First there is a preamble, which typically contains typesetting directives that apply to the entire document, as well as meta-information about the document and definitions for programming macros to be used. After that, there is the actual document which contains the text to be typeset as well as the directives for sectioning and styling the pages. In HTML these two parts are enclosed in `<head>` and `<body>` tags, respectively. In XHTML, it is also important that there be a top-level tag, for which we use `<html>`. Thus, an exceptionally simple web page might look something like this.

```
<html>
<head>
<title>Hello</title>
</head>
<body>
<p>Hello, world!</p>
</body>
</html>
```

This illustrates the essentials of HTML. Each opening tag has a closing tag; the tags enclose the text to that they format; and there are header and body divisions. Within this format, it is quite straightforward to apply other HTML formatting options using the tags described below. Note that in HTML5, the `<html>` and `<head>` tags are considered to be unnecessary.

One of the more important points about this is that modern practice in HTML is to avoid putting specific style information in an HTML page. Instead, all of the tags in the HTML file are *structural* – they pertain to the structure of the document. We try to apply styles through external specifications and customizations of the structural markup. There are tags that apply style information - such tags are called *procedural* markup. This is frowned upon, but we include a few such tags (e.g. the `<b>` tag) below. Here is a list of the simplest HTML tags.

`<b>` Encloses bold-faced text.

`<br/>` Makes a line-break. This contains no text, so it opens and closes itself.

`<div>` Encloses a structural division of the document.

`<em>` Encloses emphasized text. This usually just italicizes text.

`<h1>`, `<h2>`, ..., `<h6>` Encloses a heading at a level indicated by the integer; i.e. `<h1>` typically gives a large major heading, while `<h4>` is a smaller, more minor heading.

`<hr/>` Inserts a horizontal rule. This contains no text, so it opens and closes itself.

`<i>` Encloses italicized text.

`<ol>` Encloses an ordered (numbered) list.

`<p>` Encloses a paragraph.

`<span>` Encloses a relatively small segment of text to be formatted.

`<tt>` Encloses teletype (monospace) text.

`<ul>` Encloses an unordered (bulleted) list.

2.3.2 Links to external objects

What makes “hypertext” hyper is the ability to link immediately to other pages, to see images within the text, and to include other kinds of content directly or indirectly into the page. This requires us to be able to tell the browser how to find those external pages and images. This is done using URLs in special HTML tags. To make a link to a different web page, HTML provides the “anchor” tag:

```
<a href='http://www.whatever.edu'>This is what the user sees</a>
```

The key “href” stands for hypertext reference, and just allows you to specify the URL for the file you want to link to. The text enclosed by the `<a>` tag is what the viewer sees in her browser. Note that there is a `<link>` tag in HTML, but it does something entirely different from what we call a “link” in normal usage – use the anchor tag `<a>`.

Images are another kind of external content that appears in web pages. The image tag is somewhat similar to the anchor tag, however, for reasons that are difficult to fathom, the keyword specifying the URL of the image is different. Here is an image tag.

```
<img src='http://www.whatever.edu/myimage.jpg' alt='alternate text'/>
```

Instead of using the keyword “href”, the `<img>` tag uses “src”. Since the tag itself will give the image, it does not need to enclose text, so it can close itself, hence the slash at the end. The “alt” keyword allows you to specify text that will appear in place of the image if for some reason the image file is not found or cannot be loaded.

There are many other ways to include external content. For example, one way that extraneous input can be loaded into an HTML page is through an *iframe*. This displays a web page or other content as a page-within-a-page. The idea is to make a little frame within your web page that has its own scroll bars and such, which can display some other page you specify.

```
<iframe src='http://www.whatever.edu/otherpage.html'
    width='400' height='400'>text</iframe>
```

This creates such a frame, and the alternate text “text” appears if the browser does not support iframes. It is necessary to specify the width and height of the frame (in pixels), so that the browser knows how to place it. IFrames are used more often in conjunction with forms that users can fill out than as static elements of a page.

Table 2.2: Simple HTML Table

First	Row
Second	Row

2.3.3 Tables

Typesetting in HTML is so limited that alignments of any kind are quite difficult. From the beginning, alignments HTML required the use of tables. In this century CSS allows us to do some moderately sophisticated alignment, but tables are still required for many details. Fortunately these are fairly simple in HTML.

To create a table in HTML, simply use the <table> tag. This can take arguments for width and height, or even some rudimentary aspects of borders and padding. Ordinarily most of those specifications are handled through CSS, which we will discuss later. The <table> tag does nothing but create a space for a table. In order to fill it with rows and columns, we must include <tr> (table row) and <td> (table data) tags. It is important to note that tables are entered row by row, and the only indication of columns takes place through the <td> tags.

To see how this works, look at the HTML fragment below.

```
<table>
<tr><td>First</td><td>Row</td></tr>
<tr><td>Second</td><td>Row</td></tr>
</table>
```

This table has two rows (indicated by <tr>) and two columns (essentially indicated by <td>). It produces output similar to that in Table 2.2. Note that it does not put any borders on the table, and it might not align the elements the way you want. We can do all these things, but they are formatting, which we will discuss at length in the CSS section.

There are two formatting options that we should discuss here instead of waiting for the style section. Typically, the first row of a table contains column headings, which should be formatted differently from the rest of the entries. For that purpose HTML provides a <th> (table header) tag. By default this tag puts the table headers in a bold faced font. Again, sometimes one needs a row of the table that spans more than one column. The <td> and <th> tags provide an option for that, called colspan. We can choose the <td> that represents the first cell/column that will be combined, and in that tag append the option e.g. colspan='2' to cause the content of that cell to span two columns. Here is a modification to our little table from earlier to provide a sort of title inside the table. The results are illustrated in Table 2.3.

```
<table>
<tr><th colspan='2'>A Simple Table</th></tr>
<tr><td>First</td><td>Row</td></tr>
```

Table 2.3: Simple HTML Table With Header

A Simple Table	
First	Row
Second	Row

```
<tr><td>Second</td><td>Row</td></tr>
</table>
```

Note that tables appear in HTML much more often than you might think. Any time one needs a multicolumn alignment, or navigation, or specific placement of images, or careful organization of headers, an HTML table is probably involved. In our examples we saw no borders, and this illustrates the way tables are used in HTML very commonly for every kind of text alignment imaginable.

2.4 Cascading Style Sheets

As the World Wide Web has matured, people have demanded more of it. Establishments doing business over the Web find themselves managing sites containing thousands or tens of thousands of separate pages, each of which requires formatting for presentation. These companies, universities, and government agencies want a uniform appearance for their pages, and want to be able to change that appearance at will. They need to create pages with the standardized style in a short time, and want to have complete control over the appearance of their pages.

In 1997, the W3C promulgated a proposed standard in response to this need. The standard was for Cascading Style Sheets (CSS). These are HTML pages that contain only formatting instructions. The idea is that we should be able to style our headings, text and other HTML constructs in any way we like without having to type those details in every single tag. Moreover, those style specifications should reside in a single file that all other HTML files might then refer to. In that case, a change to the style file would simultaneously change the presentation of every page that used it.

For example, suppose that we want our principal heading always to appear in sans serif font, in our corporate color, centered on the page. Years ago it was common to put a font format everywhere we wanted to customize the appearance of text.

```
<h1 align="center">
<font color="#006666" type="Arial,Helvetica">
Our heading text
</font></h1>
```

There are several problems with this approach. Evidently it requires a good deal more typing. It requires us to put in an extra tag with the font specification every time a heading appears. Worse yet, the <font> tag shows nothing about the structure of the document, so it is in some sense extraneous to the goals of the page. Finally, if the institution changes its Web standards next week, then we might have to change those tags on each of the ten thousand pages we maintain to the following set of tags:

```
<h1 align="right">
<font color="#0000A0" type="Times">
Our heading text
</font></h1>
```

Making these changes and others like them would require many person-weeks of work, thus adding to the cost of the acquisition. Style tags address this issue by allowing us to set the formatting specification for any HTML tag in a single place. In the above example, we might put instructions at the top of each page. They would look like this:

```
<style type="text/css">
h1 {
 text-align:  center;
 color:  #0000A0;
 font-family:  Times New Roman, Times, serif;
}
</style>
```

All of our heading lines would then look much simpler: `<h1>Our heading text</h1>`. Now the tag shown above is only valid for the page in which it appears. We would prefer by far to be able to set styles across a large number of pages. We need a means of having one or more pages that define the styles for our site, which may be loaded from any page on that site. In short, we need the "link" tag. In the above example, we would put the style specification above in some file, say "company_style.css". The "css" extension is standard for a style sheet. After that, we could gain access to the styles defined in that file by putting the following tag in all of our HTML files:

```
<link rel='stylesheet' type='text/css' href='company\_style.css' title='main\_style'>
```

The link tag is largely self-explanatory. The critical item is the `href` attribute, which gives the URL of the stylesheet. The `rel` attribute stands for "relation" – the relationship the linked file holds to the current page. The `title` attribute permits us to name more than one style sheet that applies to any given document. The use of multiple stylesheets is the reason for the term "cascading" that appears here. Each subsequent style sheet adds to the one before, or replaces its attributes, if there are duplicates. The basic format for styles is simple: there are attributes that apply to individual HTML tags, and we enclose the attributes within braces, separated by semicolons. In abstract, style specifications have the following form.

```
tag {
attribute 1:  value 1;
attribute 2:  value 2;
...
attribute n:  value n;
}
```

The colon separating the attribute from the value is obligatory, as is the semicolon at the end of the line. The tabs at the beginning of each attribute line only contribute to the readability of the specification, and are optional. Thus, we could specify a particular paragraph style as follows.

```
p {
font-family:  sans-serif;
font-size:  larger;
color:  #a00000;
text-align:  center;
border:  dashed;
}
```

This would force every paragraph in the document to be displayed in some kind of sans-serif font, in a larger size than the default, in a dark red color. The text would be centered on the page, with a dashed border around it. An exhaustive list of attributes may be found at the W3C site: `http://www.w3.org/`. We also have an abbreviated listing of attributes in Section 2.4.1.

2.4.1 Style Sheet Properties

background-color – Possible values include a color specification or "transparent". This attribute applies to all elements, and is not inherited.

background-image – Possible values include a URL for the image, or "none". This attribute applies to all elements, and is not inherited.

border – This attribute is used to set and color border around any element. It is a shorthand notation encompassing the attributes border-top, border-right, etc. Possible values include a width, a style, or a color. The width could be a length or a percentage. The style specification could be "dotted", "dashed", "solid", or one of several other choices. Colors are specified as usual.

color – The value must be a color specification, as in "ff0000" for red, or "a000a0" for a purple. It applies to all elements. This attribute may be inherited. These specifications may also be made in the form rgb(i,j,k), where i, j, and k represent numbers between 0 and 255, inclusive. The number i again denotes the amount of red in the color, j the amount of green, and k the amount of blue.

float – Possible values include “left”, “right”, and “center”. This is the means of aligning non-text elements on the page. It converts the current element into a “floating element”, which disconnects it from the flow of text on the page. For example, if a block were made to float left, then it would start in its assigned place in the text, but other subsequent text would be allowed to flow around the floating element – think of a picture that is aligned left with the text flowing around it to its right.

font-family – Possible values include a specific family, such as “Arial” for Microsoft machines, or a generic family, such as “sans-serif”. More than one font family may be specified, separated by commas.
The browser is then to look through its list of fonts for one that fits one of the members of the list. Thus “Arial, Helvetica” would allow a browser on a Unix computer to look for an Arial font, fail, and then find a Helvetica font. Both fonts would satisfy the generic appellation ”sans-serif”. There are only a few generic families defined: “serif”, “sans-serif”, “cursive”, “fantasy”, and “monospace”.

font-size – Possible values include absolute sizes (such as “x-small”, “small”, “medium”, “large”, or “x-large”), or relative sizes (such as “smaller” or “larger”). It is also possible to specify the font size in an actual unit of length or percentage of screen size, e.g. `10px` or `14pt`.

font-style – Possible values include “normal”, “italic”, and “oblique”. It applies to all elements, and is inherited.

font-weight – Possible values include “normal”, “bold”, and “bolder”. There are other settings in the form of numbers from 100 to 900. This attribute applies to all elements and is inherited. The attribute applies to all elements and is inherited.

height – The value must be a length. It applies to block-level elements, and is not inherited.

length units – There are many systems of units that you may use to specify length measurements. These include absolute units such as “px” - pixels; “pt” - typesetting points; “in” - inches; “mm” - millimeters; and “pc” - picas, where 1pc = 12pt. These absolute measurements should only be used when the physical characteristics of the output device are known - i.e for printing styles. For screen styles, one should use relative measurements such as “em” - the width the the letter M in the current font; or “ex” - the height of the letter x. Alternatively one might use keyword values such as “x-small”, “small”, “medium”, “large”, “x-large”.

margin – Possible values length units or percentage units. The margin specification is the width of the space *outside* the current element but inside the page. The margin value applies to all four sides of the object in question: top, bottom, left, and right. If you need to set only one of these, you may use e.g. the “margin-left” attribute. This attribute applies to all elements and is not inherited.

padding – Possible values length units or percentage units. The padding specification refers to white space *inside* the current element around the text it encloses. The padding value applies to all four sides of the object in question: top, bottom, left, and right. If you need to set only one of these, you may use e.g. the “padding-left” attribute. This attribute applies to all elements and is not inherited.

percentage units – these always refer to some reference unit such as the width of the screen, or a standard font size. These allow relative specifications for all lengths with high precision. The format is always a number followed by a percent sign (%).

text-align – This sets the alignment of text within the current element. Possible values include “left”, “right”, “center”, and “justify”. It applies to block-level elements. This attribute may be inherited.

width – The value could be a length or a percentage of the width of the screen. This attribute applies to block-level elements. It is not inherited.

2.4.2 Style Classes

You will at some point want to have various different styles for text in your document. For example, you might want to emulate this book, and set most text in a larger sans-serif font, but have certain examples or illustrations in a smaller font, enclosed within a border. In short, the single paragraph specification above would not be sufficient. Instead, you would create several different paragraph styles. You do this by appending a period to the name of the element for which you want several styles, followed by names of the styles. For example, suppose we want a default paragraph style in a larger sized sans-serif font, but also want a second style in a smaller sans-serif font colored blue. The following style specification would do this.

```
<style type="text/css">
p {
font-family: sans-serif;
font-size: larger;
}
p.smallblue {
font-size: smaller;
color: #0000a0;
}
</style>
</p>
<p>This is in the larger font.
```

```
<p class="smallblue">This is the blue font, that actually
appears in the default size.
We see that if we want the larger font, we simply use the
'p' tag, but if we want the smaller blue font, we
use 'p class="smallblue"'.  Everything that is described in
the "smallblue" paragraph style is relative to
the parent paragraph style.
Thus, the font for normal paragraphs is
larger than the standard, while that
for the "smallblue" paragraphs is actually the standard size.
The "smallblue" paragraphs also receive
their font family from the standard paragraph, so that they
appear in a sans-serif font.</p>
```

In short, we may specify as many different styles for any element as we wish, referring to them using the class attribute. The attributes of the parent element style are kept by the subordinate element styles, unless those attributes are set explicitly in a style declaration. For example, if we add a declaration of the form `p.serif font-family:  serif;` to our style sheet, we have a new paragraph style using a serif font, in spite of the setting in the parent paragraph style.

We can define classes that apply to all possible HTML tags by omitting the tag name. For example, we could have defined the 'smallblue' class above to apply not only to paragraphs, but also to list items, headings, or any other tag that formats text by specifying it as follows.

```
<style type="text/css">
.smallblue {
font-size: smaller;
color: #0000a0;
}
</style>
```

The simple omission of the tag name causes 'smallblue' to be a generic formatting attribute of an HTML element for which it makes sense.

2.4.3 Style IDs

There is another way to customize styles according to specific uses. We can use ID tags to associate styles with specific segments of text. In the style file IDs are referenced using pound signs (#). Here is some HTML with two divisions that are formatted differently.

```
<style type="text/css">
p.blue {color: blue}
#special {font-style: italic}
#special p{color: red}
```

```
</style>
<div><p>This is a paragraph in the first
division, set in the default font, style, and color.</p>
<p class='blue'>This is another paragraph which is blue
because we made it a particular class.</p>
</div>
<div id='special'>
<h3>Here is the title for a new section - it is italicized</h3>
<p>This is a paragraph in the new section. It is italicized
because the division has id 'special', and it is red
because it is a paragraph in that division.</p>
```

The obvious question is why we need both IDs and classes, since their function is similar. If you look closely, however, you'll see that their function is not so similar as it first appears. Classes allow us to apply formatting options very specifically to individual instances of HTML tags. By contrast, IDs permit use to specify formats for entire sections of text, without identifying the elements within those sections. In the example above, we saw that the first paragraph was not identified as being of any class or ID, so it had a default appearance. We identified the second paragraph as class 'blue', so it takes on the attributes of that class. However, we then use a 'div' tag to create a new section of our document, and identify the entire division with the ID 'special'. This means that all text in that section is italicized, i.e. both the heading and the paragraph are italicized. We also specified that all paragraphs in a division with ID 'special' will be set in red text. Thus, the paragraph in that division appeared in red, even though it had no identifiers in the actual paragraph tag.

This brings out the point of IDs. IDs are usually associated with 'div' tags, which are purely structural markup. By dividing our documents using 'div' tags with IDs, we allow formatting to be applied section by section. Classes only allowed us to apply formatting by label. To put it another way, an ID is typically used to specify formatting for the tags used in a given section of a document. A class is used to change the formatting of a single object on a page.

Just to reinforce this, here is an illustration of some CSS specifications that occur often.

p.special – applies formatting to any paragraph specified to be of class 'special'.

#special p – applies formatting to *all* paragraphs in a division with id 'special'.

This also emphasizes that IDs are almost always applied to divisions of HTML documents. One puts IDs on containers, and the formatting associated with them applies to everything contained inside.

2.4.4 Pseudo-classes

One innovative and useful thing about CSS is new functionality it introduces through pseudo-classes. Pseudo-classes allow us to modify the behavior of

HTML elements based on their state: interaction with the user, position within a container, or position relative to other elements. This can make a page somewhat more interactive, and sometimes even easier to use.

The most common use of pseudo-classes is to change the appearance and behavior of links to other pages. We can change the color of a link based on whether the site it points to has been visited before, whether the mouse is hovering over the link, or whether it is currently in use. Here is some HTML with CSS to make the link appear black with no underline ordinarily, but when a user hovers the cursor over it, the link turns red, and if the user clicks the link, the text enlarges somewhat. After the site the link points to has be visited, the link appears in grey.

```
<style type="text/css">
a{text-decoration:none}
a:visited{color:grey}
a:hover{color:red}
a:active{color:red; font-size: larger}
</style>
<ul>
<li>Institution 1: <a href='http://wsu.edu'>Washington State University</a></li>
<li>Institution 2: <a href='http://ucdavis.edu'>University of California at Davis</a></li>
</ul>
```

We can see that we specify pseudo-classes using a colon (:). Attaching the pseudo-class to a specific HTML tag is not necessary. For example, the specification for the 'visited' pseudo-class could have been made as

```
:visited{color:grey}
```

However, pseudo-classes usually do pertain to specific HTML tags, so invoking the names of those tags is common.

There are some pseudo-classes that select HTML elements based on their relationship to a parent. In the example above we used an unordered list <ul> tag. This uses list item tags that are effectively children of the parent <ul> tag. For example, we could insert

```
li:first-child{color:yellow}
```

into the style section of the example above to cause the first list item to appear in yellow. This color does not supersede other colors applied according to the state of the link. While it is again not necessary to specify that this applies only to the <li> tag, in this case that is important. Most of the tags in the file are children of some others, so if you don't specify what HTML the child pseudo-class refers to, you might get yellow text throughout your document. Note that there is also a `:last-child` pseudo-class.

Another more interesting pseudo-class is the `:nth-child(N)` class. This can interpret simple algebraic expressions to apply formatting to selected child tags. For example, if instead of the `:first-child` pseudo-class we used the following, we would see even-numbered list items appear in yellow.

```
li:nth-child(2n){color:yellow}
```

This can be used in a number of settings. For example, in a table we could make every odd-numbered column have a yellow background by using the `:nth-child()` pseudo-class.

```
td:nth-child(2n-1){color:yellow}
```

If you are unable to specify the particular HTML element you want a pseudo-class to apply to, but need also to apply attributes only to certain among the children of an element, you can use the `:first-of-type`, `:last-of-type`, and `:nth-of-type()` pseudo-classes. These work like the child pseudo-classes, except in that they can apply to situations where more than one kind of HTML element is involved.

One other common use of pseudo-classes involves changing the appearance of the first line or even letter of a text element. The following example makes the first letter of any paragraph larger than the rest, and floated left so that text flows around it.

```
<style type="text/css">
p:first-letter{font-size: x-large; float: left}
</style>
<p>The first letter of this paragraph should be much larger than the
rest, and floated left so that the rest of the paragraph flows around
it.  This can be a nice special effect.</p>
```

There is also a `:first-line` pseudo-class. You can even combine things:

```
p:first-child:first-letter{font-size: x-large; float: left}
```

which causes only the first letter of the first paragraph in a section to have the formatting.

2.5 MathML

Mathematical typesetting is tremendously different from ordinary line-based text typesetting. For ordinary text, one picks a font and a baseline, and then just figures out how wide each character is; then from that how wide each word is; and then stretches the spaces uniformly to fill the gaps between those words. Mathematical typesetting differs from this in many ways.

- Mathematical typesetting uses a different font with different spacing between characters than ordinary typesetting;
- it is frequently not line-oriented;
- it has complex alignments both horizontally and vertically;
- it uses a huge character set;

- it has some symbols that change size and have other typesetting around them that depend on the situation.

To summarize, it is much more complicated.

Because of this complication, early HTML versions simply did not support mathematics. In the bad old days, the only reasonable way to display mathematics on the web was to convert each equation to an image and embed that in the HTML document. There were other tricks: a java interface that displayed math; various plugin programs for browsers that could do some math. None of these were very satisfactory.

In this vacuum many things were possible. Some people feared those possibilities, and in particular feared that whatever standard eventually won out for Web mathematical typesetting would preclude efficient searches or interaction with mathematical objects. For that reason the W3C (World Wide Web Consortium) became involved, and proposed MathML (Mathematical Markup Language) as a standard in 1998. MathML was originally conceived as a subset of XML (eXtensible Markup Language). XML was designed to be an extremely versatile framework in which to develop content-oriented markup, and putting MathML into this framework was easy. Unfortunately, it also made generating and supporting MathML much more difficult, especially in that mathematical documents usually mixed ordinary text with mathematical notation and graphics. In XML this meant that one had to go to extraordinary lengths to get the XHTML, MathML, and SVG (Scalable Vector Graphics, to be discussed later) to play together. The XHTML/MathML standard did not become popular. Many browsers did not even support MathML.

At the end of the first decade of this century, many people, including some at the W3C, came to realize that XML, while a nice idea, was not going to be able to handle the many different typesetting, graphic, and video formats that were inundating the Web. In order to channel development into a constructive direction, the W3C created HTML5, a new standard for HTML that incorporated video, audio, as well as MathML and SVG. Suddenly multiple browsers supported the standard.

To create MathML one usually uses some program that one can interact with graphically to choose mathematical symbols and arrange them. The program then generates the code for you. Alternatively there are programs that convert other mathematical languages such as $\LaTeX$ to MathML. In particular, there is an application called MathJax supported by the American Mathematical Society that translates LaTeX into MathML on the fly, inside a browser. Nonetheless, here we will learn the actual syntax for MathML. The fact is that currently the graphical tools are not that easy to use, and are limited in what they can do, while the conversion tools fail often. Anyone who writes a lot of mathematics will need to understand the syntax of MathML. On the other hand, that syntax is easy, although verbose.

2.5.1 MathML Syntax

MathML was designed as a small subset of XML, so its syntax is basically that of XML. The rules are simple.

1. MathML commands are tags as with HTML, enclosed in "angle brackets". MathML tags themselves enclose the text that is to be formatted.
2. There must be a top-level tag that encloses all the content. In MathML that tag is always `<math>`.
3. Every opening tag must have corresponding closing tag.
4. Tags must be nested properly.

There is a little more to it than this. The name of every MathML tag starts with "m". MathML was designed from the start to objectify mathematical notation, so MathML tags enclose mathematical notational objects. Finally, mathematical notation uses characters as its basic objects: either numbers or letters that are variable names. Thus MathML formats things by objects, but ultimately one character at a time.

A few examples will make this more clear. We are used to typing x^2 as `x^2` or `x**2`. In MathML typing this expression is significantly more troublesome, viz.

```
<math><msup><mi>x</mi><mn>2</mn></msup></math>
```

We can see many of the ideas of MathML here. First, we needed a tag to enclose all the content: that is the `<math>` tag. Next, we need to make the entire exponent structure, so we have the `<msup>` tag. That MathML command takes two arguments, i.e. between `<msup>` and `</msup>` there must be exactly two MathML objects - no more, no less. The first of those objects is taken to be the mantissa of the exponential expression, and the second is the exponent. In the example above, the first argument (the mantissa) is `<mi>x</mi>`, which represents x set in a math italic font. It is important to note that `<mi>` itself takes exactly one argument; if you put more than one character between `<mi>` and `</mi>`, then neither will have the math italic formatting applied. This leads to much tedium, but as we have noted, MathML was never really meant to be typed. The second argument to `<msup>` is `<mn>2</mn>`. Here `<mn>` stands for math number. It does not noticeably change the appearance of the "2" – there are no math italic fonts for numbers – but it handles spacing.

Obviously there is a lot of pain in moving from `x^2` to the MathML representation for that, but as we noted, the people who designed MathML wanted to be able to search and query the Web for information, so they wanted the representation of mathematics to be highly structured and meaningful in itself. When we type `x^2` it is just a string of three symbols; whereas when we have all the structure of the MathML representation, it is clear that we are talking about a mathematical exponential construction. If you should have the misfortune of typing MathML from first principles, just remember that you are helping search engines everywhere.

What if the exponent contains more than one character, as in x^{2t}? Since the `<msup>` tag requires exactly two arguments, we cannot possibly expect it to accept or understand the three objects that will appear in that expression. We need to group the characters in the exponent so they appear to the `<msup>` tag as one object. This is what the `<mrow>` tag was created for. It simply encloses any number of other MathML objects to cause them to present to some picky tag as a single object. Here is the MathML to typeset x^{2t}.

```
<math><msup><mi>x</mi>
<mrow><mn>2</mn><mi>t</mi></mrow></msup></math>
```

Fortunately MathML ignores white space characters between tags, including the newlines here. Obviously MathML expressions become very long very quickly. Note that superscripts are handled by the `<msup>` tag as we have shown. We can do subscripts in exactly the same way using the `<msub>` tag.

Typically mathematical expressions are much more complicated than the ones we have typeset so far. Often these expressions are “displayed”: they are typeset outside the line of text, in their own area, with perhaps larger symbols or an expanded format. To do this in MathML requires adding an attribute to the `<math>` tag.

```
<math mode='display'>
<math><msup><mi>x</mi><mn>2</mn></msup></math>
</math>
```

This command typesets

$$x^2.$$

The only addition required in this case is the `mode='display'` in the `<math>` tag.

Usually when we display mathematical expressions, they are more complicated than what we have done so far. Consider the equation

$$f(t) = \left(\frac{t}{2}\right) e^{t^2/4}.$$

We can typeset this using MathML, but the code is getting longer.

```
<math mode='display'>
<mi>f</mi><mfenced><mi>x</mi></mfenced><mo>=</mo>
<mfenced><mfrac><mi>t</mi><mn>2</mn></mfrac></mfenced>
<msup><mi>e</mi>
<mrow><msup><mi>t</mi><mn>2</mn></msup><mo>/</mo><mn>2</mn></mrow>
</msup>
</math>
```

There are three new tags here: `<mfenced>`, `<mfrac>`, and `<mo>`. The first puts parentheses around its contents, and the second creates a fraction. The `<mo>` tag is simplest. It simply formats symbols properly for mathematics. Equals signs need a certain amount of space around them - `<mo>` provides that. Minus signs are wider than hyphens, `<mo>` makes that change. In general, you will always want `<mo>` tags around symbols that are neither variables nor numbers. The `<mo>` tag takes exactly one argument.

The `<mfenced>` tag can take any number of arguments, but it separates arguments with commas. The reason is that it is designed to display vectors or points in n-space. This makes typesetting a point $(2, 2, -2)$ relatively easy:

```
<math><mfenced>
<mn>2</mn><mn>2</mn><mrow><mo>-</mo><mn>2</mn></mrow>
</mfenced></math>
```

Note that we did have to use an `<mrow>` around the last entry because it actually involved two characters – we did not want a comma between the minus sign and the 2. The best thing about `<mfenced>` is that it can decide how big the parentheses should be. It looks at the content inside the tags, computes how tall that is, and stretches the parentheses to enclose that content properly. Thus the parentheses on the left side of the equation are small, while those enclosing $t/2$ are large.

The `<mfenced>` tag can produce any kind of grouping symbol you like - parentheses are only the default. For example, if you want to enclose an expression in brackets, you can use the `open` and `close` attributes. Here is an expression in brackets.

```
<math>
<msup>
<mfenced open='[' close=']'>
<msup><mi>x</mi><mn>2</mn></msup><mo>+</mo><mn>2</mn>
</mfenced>
<mrow><mn>1</mn><mo>/</mo><mn>2</mn></mrow>
</msup>
</math>
```

This produces $\left[x^2 + 2\right]^{1/2}$. Likewise, one can use `<mfenced>` to enclose things in braces, or by making the close empty, one can use it to place a brace only on the left, as in a piecewise function definition.

The `<mfrac>` command requires exactly two arguments: the numerator and denominator of the fraction, respectively. Obviously we can use `<mrow>` tags as needed to fit things in.

Some mathematical constructions require relatively complex alignments that can even change depending on whether an equation is displayed or not. One example of such a construction is the summation symbol. When we typeset a summation in a line of text, the starting and ending indices are displayed as subscript and superscript. For example, examine $\sum_{n=0}^{\infty} a_n x^n$. When we display that same summation, the starting and ending indices appear under and over the summation symbol, as with

$$\sum_{n=0}^{\infty} a_n x^n.$$

Notice that the summation symbol itself gets bigger in the displayed expression.

In MathML we have the means to typeset each of these forms. Here is the code for the first form.

```
<math>
<msubsup>
<mo>&sum;</mo>
<mrow><mi>n</mi><mo>=</mo><mn>0</mn></mrow>
<mo>&infin;</mo>
</msubsup>
<msub><mi>a</mi><mi>n</mi></msub>
<msup><mi>x</mi><mi>n</mi></msup>
</math>
```

We have introduced a new object here. The summation symbol is created as `&sum;` – both the ampersand that starts it and the semicolon that ends it are important. This summation symbol has an important property: it can change size. The `<mo>` tag takes care of determining how big the summation symbol should be.

We created the summation with the subscripts and superscripts using the `<msubsup>`. Evidently this takes three arguments: the central symbol, the subscript, and the superscript, in that order. We put each argument on a separate line, and obviously the bottom index contained three symbols, so we had to enclose it in an `<mrow>`. The rest of the expression only requires simple applications of `<msub>` and `<msup>`.

If we wanted to display that same expression, we have to change the mode, and change the tag used for the summation construction to `<munderover>`.

```
<math mode='display'>
<munderover>
<mo>&sum;</mo>
<mrow><mi>n</mi><mo>=</mo><mn>0</mn></mrow>
<mo>&infin;</mo>
</munderover>
<msub><mi>a</mi><mi>n</mi></msub>
<msup><mi>x</mi><mi>n</mi></msup>
</math>
```

Everything else is just the same. Just as with `<msubsup>`, `<munderover>` requires three arguments.

There are certain symbols besides fractions in mathematics that require aligning objects atop or beneath one another. For example, the limit notation requires the fragment “lim” to have an indication directly underneath it concerning what variable approaches what constant.

$$\lim_{n\to\infty} \frac{1}{n} = 0.$$

We can do this in MathML using `<munder>`. This tag requires two arguments: the main text, and the smaller text that appears beneath it. Here is the code.

```
<math mode='display'>
<munder>
<mtext>lim</mtext>
<mrow><mi>n</mi><mo>&rightarrow;</mo><mo>&infin;</mo></mrow>
</munder>
<mo>=</mo><mn>0</mn>
</math>
```

Again, sometimes you might need to form a new mathematical object by putting one symbol over another. For example, one way to annotate vectors uses letters with a small arrow atop them, as with $\vec{v} = (v_1, v_2)$. We can do this in MathML using `<mover>`.

```
<math mode='display'>
<mover>
<mi>v</mi>
<mo>&RightArrow;</mo>
</mover>
<mo>=</mo>
<mfenced>
<msub><mi>v</mi><mn>1</mn></msub><msub><mi>v</mi><mn>2</mn></msub>
</mfenced>
</math>
```

There are a couple of other tags worth mentioning here. They both have to do the the root symbol. The `<msqrt>` tag does what it looks like it should: it encloses its single argument in and under a square root symbol, as with $\sqrt{x^2+2}$. This example can be done in MathML as follows.

```
<math>
<msqrt><mrow>
<msup><mi>x</mi><mn>2</mn></msup><mo>+</mo><mn>2</mn>
</mrow></msqrt>
</math>
```

If you are making e.g. a cube root instead of a square root, you want a slightly different notation, as in $\sqrt[3]{x^2+2}$. This uses the `<mroot>` tag, which takes two arguments: the quantity whose root you need, and the actual root. Here is the code.

```
<math>
<mroot><mrow>
<msup><mi>x</mi><mn>2</mn></msup><mo>+</mo><mn>2</mn>
</mrow>
<mn>3</mn>
</mroot>
</math>
```

2.5.2 MathML Alignments

It is very common in mathematics to impose various kinds of alignment on the symbols and expressions that appear. Matrices, chains of equations, piecewise function definitions, and various other constructions all require fairly complicated alignments. In MathML these alignments are all done using tables that work very much like tables in HTML.

The basic tag to make a table in MathML is <mtable>. It operates very like the corresponding tag in HTML: rows are enclosed by <mtr> tags, and cells within those rows (which correspond to entries by column) are enclosed in <mtd> tags. With these simple tools, it is an easy matter to make matrices. We can typeset the matrix

$$A = \begin{pmatrix} 1 & 2 \\ 0 & 1 \end{pmatrix}$$

using the MathML code below.

```
<math mode='display'>
<mi>A</mi><mo>=</mo>
<mfenced>
<mtable>
<mtr><mtd>1</mtd><mtd>2</mtd></mtr>
<mtr><mtd>0</mtd><mtd>1</mtd></mtr>
</mtable>
</mfenced>
</math>
```

Technically we might have done well to enclose the numbers in the matrix in <mn> tags, but those have more to do with spacing than appearance, and this worked well enough without the extra typing.

As we noted in the previous section, it would be easy to change the parentheses surrounding the matrix to brackets. Only one line would change: the <mfenced> tag would change to

```
<mfenced open='[' close=']'>
```

to get the the display

$$A = \begin{bmatrix} 1 & 2 \\ 0 & 1 \end{bmatrix}$$

Reference 2.4

MathML Tags

mfenced – Math fenced content. This takes any number of arguments, but separates them with commas, as for point or vector coordinates. If you want simply to enclose an expression in variable-size parentheses, you must enclose the expression in `<mrow>` tags.

mi – Math italic. This takes only one argument, i.e. one letter.

mn – Math number.

mo – Math object. This takes only one argument, i.e. one symbol. It manages the size of the symbol, and the space around it.

mover – Math over. This takes two arguments: the main object; and the one to be rendered above and smaller than that object.

mroot – Math root. This takes two arguments: the first is the content to go under the root sign, and the second is the root taken. E.g. if you mean to represent a cube root, then the second argument is `<mn>3</mn>`.

mrow – Math row. This is just a grouping tag. It can take any number of arguments, and causes them to be seen by any parent tag as a single object.

msqrt – Math square root. This takes only one argument, and puts that argument under a square root sign.

msub – Math subscript. This takes two arguments. The first is the object to receive the subscript; the second is the subscript itself.

msup – Math superscript. This takes two arguments. The first is the object to receive the superscript; the second is the superscript itself.

msubsup – Math subscript and superscript. This takes three arguments: the object to be sub- and superscripted; next the subscript; and finally the superscript.

mtext – Math ordinary text. This takes any number of characters of input and renders them in an ordinary (not math italic) font.

munder – Math under. This takes two arguments: the main object; and the object to be rendered underneath and smaller than the first. Use this e.g. for integral signs.

munderover – Math under and over. This takes three arguments: the main object; the one to be rendered under and smaller than that; and the one to appear above and smaller than the main object. Use this e.g. for summation symbols.

A similar kind of alignment occurs when defining a function piecewise. For example, the function definition

$$p(x) = \left\{ \begin{array}{ll} 0 & x < 0 \\ x^2 & x \geq 0 \end{array} \right.$$

involves two left-aligned columns with the brace only on the left. One way to do this in MathML goes like this.

```
<math mode='display'>
<mi>p</mi><mfenced>x</mfenced><mo>=</mo>
<mfenced open='{' close=''>
<mtable>
<mtr><mtd><mn>0</mn></mtd>
  <mtd><mi>x</mi><mo>&lt;</mo><mn>0</mn></mtd></mtr>
<mtr><mtd><msup><mi>x</mi><mn>2</mn></msup></mtd>
  <mtd><mi>x</mi><mo>&ge;</mo><mn>0</mn></mtd></mtr>
</mtable>
</mfenced>
</math>
```

Another place where alignments come up commonly in mathematics is in chains of equations. Here is one such alignment, albeit a particularly simple one:

$$\begin{aligned} d(x) &= x^2 + 2x + 1 \\ &= (x+1)^2 \end{aligned}$$

This is actually quite troublesome to do in MathML.

```
<style>
.lhs{text-align:right;padding-right:0px}
.rhs{text-align:left;padding-left:0px}
.c{text-align:center; padding: 0px}
</style>
<math mode='display'>
<mtable>
<mtr><mtd class='lhs'>
<mi>d</mi><mfenced><mi>x</mi></mfenced>
</mtd><mtd class='c'>
<mo>=</mo>
</mtd><mtd class='rhs'>
<msup><mi>x</mi><mn>2</mn></msup>
<mo>+</mo><mn>2</mn><mi>x</mi><mo>+</mo><mn>1</mn>
</mtd></mtr>
<mtr><mtd>
</mtd><mtd class='c'>
<mo>=</mo>
```

```
</mtd><mtd class='rhs'>
<msup>
<mfenced><mrow><mi>x</mi><mo>+</mo><mn>1</mn></mrow></mfenced>
<mn>2</mn>
</msup>
</mtd></mtr>
</mtable>
</math>
```

If one leaves off the extra classes defined in the style section, then there is extra padding around the cells of the table that make the equations appear too spread out. Thus, we define some classes that make left-hand-sides of equations push to the right part of their table cells, with no padding; take all the padding away from the equal signs in the middle of the alignment; and make right-hand-sides push left with no padding. Note that the first cell in the second row of the table contains the (empty) left-hand-side of the second equation. Because it is empty, we did not bother to apply the formatting class.

2.6 SVG

Scalable Vector Graphics (SVG) constitutes a standard promulgated by the W3C for Web graphics since 1999. It started as a graphic representation in the XML framework, with the idea that graphic images could be archived and searched as text, in addition to scaling well. However, late in the first decade of the twenty-first century, it was incorporated into the HTML5 standard. As with MathML, this made it much more appealing to developers. In particular, it allows us to generate graphical images at run time.

SVG is particularly useful to mathematicians seeking to develop graphics for the Web because it provides for coordinate transformations. We can type simple commands to describe the mathematical object we want to depict, using the natural coordinate system, and SVG can render that as a graphic image.

In order to illustrate the utility and relative simplicity of SVG, we'll try to recreate the image shown in Figure 2.1. The figure illustrates the approximation of the integral of the function $f(x) = x^2$ over the interval $[0, 1]$ using a composite trapezoidal rule on three subintervals.

The first thing that has to happen for our SVG diagram is to make a drawing area. This is easily accomplished using the <svg> tag. SVG is just a subset of HTML5, so the tag format will be very familiar. Our drawing area is 460 pixels square. Recall that a pixel is basically a "screen dot".

```
<svg width="460" height="460">
</svg>
```

Now the area we have created has a natural coordinate system whose origin is at the upper right corner of the area. The positive x direction is to the right, but the positive y direction of the natural coordinates is downward. This is

Reference 2.5

Math Symbols

There are many math symbols available, most of which have special names. These names all start with an ampersand, then the actual name of the symbol, followed by a semicolon. Here are a few such symbols.

&alpha; – greek alpha. All greek letters can be done this way: type the ampersand, followed by the typed-out name of the letter, followed by the semicolon.

&Alpha; – greek capital alpha. Other greek capital letters are done the same way.

&ge; – greater than or equal

&gt; – greater than

&infin; – infinity

&Integral; – integral symbol

&le; – less than or equal

&lt; – less than

&nbsp; – non-breaking space

&ne; – not equal

&RightArrow; – a right "goes to" arrow, as in "the limit as x *goes to* ∞ is . . . "

&subset; – subset

&sum; – summation symbol. Always use this for summations instead of a sigma – this symbol can change size as required.

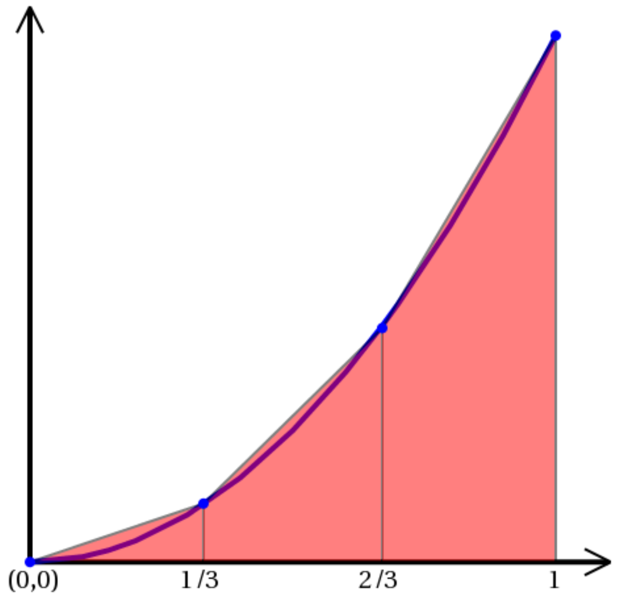


Figure 2.1: An SVG illustration of the use of a composite trapezoidal rule to approximate the integral of x^2.

the reverse of what we are used to in mathematics. It would be very useful for us to be able to plot this function in genuine Cartesian coordinates – the kind we have always used in our math classes. Therefore, we create a transformation of the natural system into a proper Cartesian system that will correspond to the axes we want. We put the commands between the <arg1> tags we already put in. The SVG we need is

```
<g transform="translate(18,425) scale(400,-400)">
</g>
```

The <g> tag is a grouping element. It allows us to apply transformations to all the SVG elements inside the group. We will put our plotting commands inside the <g> group, and they will all be interpreted in our transformed (Cartesian) coordinate system.

The transformation comprises a translation and a scaling operation. The `translate` command moves the origin of the coordinate system from the natural one (in the upper left corner of the drawing area) to the point $(18, 425)$ in the natural coordinate system. This puts the origin of the coordinates for the commands in our group in the lower left part of the drawing area, but still 18 pixels from the left, and 35 pixels from the bottom. The `scale` command tells how many pixels from the natural coordinate system will go into one unit of our scaled coordinate system. In our case we are saying that 400 pixels to the right will correspond to a distance of 1 in our transformed coordinates; and that 400 pixels in the negative y direction (up, in the natural coordinates) will correspond to 1 in the transformed coordinates.

Now all we have to do is draw our figure in our proper mathematical coordinates. First we need axes. The following commands go between the <g> tags we just put in.

```
<line x1="0" y1="0" x2="1.1" y2="0"/>
<line x1="0" y1="0" x2="0" y2="1.05"/>
```

The `<line>` tag draws a line from a point with coordinates `(x1,y1)` to another point with coordinates `(x2,y2)`. We see here that the first `<line>` tag makes a horizontal line from $(0,0)$ to $(1.1,0)$ – a depiction of an x axis. The second `<line>` tag makes a line from $(0,0)$ to $(0,1.05)$ – a vertical segment. Note that because the `<line>` tags contain no text, we make them self-closing by putting a slash before the end of the tags. Again, this is unnecessary in HTML but keeps our code consistent with XHTML.

We would like some little arrows on our axes, to show the direction of increasing x and y. One way to put those in is to use the `<path>` tag. These lines go anywhere between the <g> tags.

```
<path d="M1.105,0 l-.05,.025 M1.105,0 l-.05,-.025"/>
<path d="M0,1.055 l-.025,-.05 M0,1.055 l.025,-.05"/>
```

These are deceptively complex commands. The first puts an arrow on the x axis; the second does the arrow for y axis. We'll discuss only the first command. The `d` attribute tells the `<path>` tag that drawing commands are coming up. Inside the quotes are the drawing commands themselves. First, the capital "M" tells the path to *move* to the point given by the pair of coordinates following it: $(1.105,0)$. This means we will actually start drawing from that point – no marks have been made yet. Next, the "l" tells the path to make a *line* from that point a distance -.05 in the x direction and .025 in the y direction. In other words, it is making a short line that goes up and left a bit.

It is worth emphasizing here that in the drawing commands a capital letter indicates that the path should move or draw to an *absolute* position – a position in the coordinates of the drawing area; while a lower-case letter indicates that the moving or drawing is *relative* to the current position of the pen. Thus, the "M" moved the pen to the point $(1.105,0)$, while the "l" command said to draw a little bit left and up. We could have accomplished the same drawing using a path command `L1.00,0.025`, giving the absolute position of the end of the line. This makes the top stroke of the arrow on the x axis.

After that the remaining drawing commands are `M1.105,0 l-.05,-.025`. We now understand that to mean that we pick up our pen and move it back to the point $(1.105,0)$, and then draw a line a little bit left and downwards. This makes the bottom stroke of the arrow. The arrow on the y axis works analogously.

Next, we put in the curve for $f(x) = x^2$. We just compute a few points and plot them using the `polyline` tag.

```
<polyline points="0,0 .1,.01 .15,.0225 .2,.04 .3,.09 .4,.16 .5,.25
 .6,.36 .7,.49 .8,.64 .9,.81 1,1"/>
```

We can see pretty easily that the polyline just connects dots whose coordinates are given as ordered pairs separated by spaces in the "points" attribute. The first point in our list is $(0, 0)$; the coordinates for the next point in the list are separated from that by a space, so we see that the next point is $(.1, .01)$. The next is $(.15, .0225)$, from which we see that the points do not need to be spaced equally.

Now that the curve is displayed, we should put in the trapezoids we will use to approximate the area under the curve. This is also done using the `polyline` command. This time we make the points so as to form a trapezoid.

```
<polyline points="0,0 .33,0 .33,.1111" class="box"/>
<polyline points=".33,0 .67,0 .67,.4444 .33,.1111" class="box"/>
<polyline points=".67,0 1,0 1,1 .67,.4444" class="box"/>
```

Note that we put all these polylines in a CSS class called "box". This is so that we can shade the interior of the regions these polylines enclose. We'll do that later. Incidentally, note that we would ordinarily have made these using the <`polygon`> tag, but we wanted to illustrate the use of a CSS class to set appearance attributes for many objects of the same type.

We have points displaying the places where the edges of the trapezoidal subintervals meet the curve. We make those as small filled circles.

```
<circle cx="0" cy="0" r=".005"/>
<circle cx=".33" cy=".1111" r=".005"/>
<circle cx=".67" cy=".4444" r=".005"/>
<circle cx="1" cy="1" r=".005"/>
```

We see immediately that the circle is characterized by its center attributes `(cx,cy)` and radius attribute `r`. We did not say anything about the fill color for the circles. That is because we will do that using CSS again.

Finally, we need to put in the labels for the x axis that show the edges of the trapezoidal subintervals. This is slightly complicated by our coordinate system. Remember that we made the scale factor in the y direction negative so that our y coordinate would increase upward. Unfortunately, the letters that SVG uses for text rely on the "natural" coordinate system with y increasing downward. Thus, if we use the same coordinates for our text as for our plot, then our numbers will be upside down. We need a different coordinate system for our text – one with y increasing downward. Here is the SVG for the text labels.

```
<g transform="translate(20,425) scale(400,400)">
<text x="-.050" y=".05">(0,0)</text>
<text x=".28" y=".05">1/3</text>
```

```
<text x=".62" y=".05">2/3</text>
<text x=".98" y=".05">1</text>
</g>
```

The first thing we can notice is that we shifted the origin for our coordinate system almost as we did before. We moved it down just a little, to give some space between the numbers and the x axis. The scaling ratio of $400 : 1$ is also the same. The only significant difference is that both scaling ratios are positive this time, which means that y increases in the same direction as for the natural coordinates: downward.

The <text> tags themselves are pretty intuitive. The x and y attributes assign the upper left corner of the text position. The <text> tag encloses the text to be placed.

The last thing we need to do is color the trapezoids, set the width of the line strokes, and the color of the circles that make the dots. In a <style> area, we start by setting the color and width of the axis lines.

```
line {stroke:black; stroke-width:.01;}
```

Remember that the stroke-width is set in our transformed coordinates. We need also to do the path and polyline colors and widths.

```
path {stroke:black; stroke-width:.005;}
polyline{fill:transparent; stroke: blue; stroke-width: .01;}
```

The path was for the arrowheads, and the polylines were for both the curve itself, and the trapezoids. This should make us think for a minute: the curve looks purple and thick, while the trapezoids seem gray and thin. And we made the polyline blue, but the curve appears purple. How does that work?

The answer is that we made the trapezoids all live in a CSS class called "box". The specification for that class looks like this.

```
.box{fill: red; opacity: .5; stroke-width: .005; stroke:black;}
```

This fills the trapezoids with a red color, but it is see-through red: it is only 50% opaque. The lines are thin, and not gray but black. We drew the polyline for the curve before the trapezoids, so the translucent red fill for the trapezoids drew over the blue polyline for the curve, making it purple. The red fill also drew over the trapezoidal lines, making them lighter. The point is that the order in which we draw things matters. We could have made the polyline for the curve *after* the trapezoids, and then it would have drawn an opaque blue on top of the red fill, so we would have seen blue. However, that would not have provided an illustration of how order matters.

Finally, the CSS for the text entries is just the following.

```
text{font-size: .05;}
```

Remember that the units must be given in our transformed coordinate system.

The entire collection of SVG and CSS may be seen in Figure 2.2. You will see that making this simple picture is rather complicated in the end. This is not something we want to do manually, in general. The real point here is that we can write code that will generate this picture for us on the fly, based on the results of some scientific calculations we might have done. We'll see how to do this after we learn about programming.

Reference 2.6

SVG Tags

Here is a short list of useful SVG tags, and some of the obvious attributes that characterize them. There are many attributes for each of these tags to set appearance, but remember that this should probably be done using CSS. Do note that the CSS attributes that control line appearance are `stroke, stroke-width, and stroke-dasharray`.

<`circle`> – Makes a circle with center given in `cx` and `cy` attributes, with radius given by attribute `r`.

<`ellipse`> – Makes an ellipse with center given in `cx` and `cy` attributes, and horizontal and vertical radii given using attributes `rx` and `ry`.

<`line`> – Makes a line segment from a point given in `x1` and `y1` attributes to a point given using attributes `x2` and `y2`.

<`polygon`> – Creates a polygon with vertices given in ordered pairs separated by spaces in the `points` attribute.

<`polyline`> – Creates a chain of line segments joining points given in ordered pairs separated by spaces in the `points` attribute.

<`path`> – DRaws a (possibly discontinuous) chain of line segments that follow a complex path defined in the `d` attribute. Look online for details.

<`rect`> – Draws a rectangle with upper left corner at a point given by `x` and `y` attributes, and width and height given in `width` and `height` attributes.

<`text`> – Makes text with upper left corner at a point given by `x` and `y` attributes.

```
<style>
line {stroke:black; stroke-width:.01;}
path {stroke:black; stroke-width:.005;}
circle{fill:blue; stroke: blue; stroke-width:.01;}
polyline{fill:transparent; stroke: blue; stroke-width: .01;}
.box{fill: red; opacity: .5; stroke-width: .005; stroke:black;}
text{font-size: .05;}
</style>

<svg width="460" height="460">
<g transform="translate(18,425) scale(400,-400)">
<line x1="0" y1="0" x2="1.1" y2="0"/>
<line x1="0" y1="0" x2="0" y2="1.05"/>
<polyline points="0,0 .1,.01 .15,.0225 .2,.04 .3,.09 .4,.16 .5,.25
 .6,.36 .7,.49 .8,.64 .9,.81 1,1"/>
<polyline points="0,0 .33,0 .33,.1111" class="box"/>
<polyline points=".33,0 .67,0 .67,.4444 .33,.1111" class="box"/>
<polyline points=".67,0 1,0 1,1 .67,.4444" class="box"/>
<circle cx="0" cy="0" r=".005"/>
<circle cx=".33" cy=".1111" r=".005"/>
<circle cx=".67" cy=".4444" r=".005"/>
<circle cx="1" cy="1" r=".005"/>
<path d="M1.105,0 l-.05,.025 M1.105,0 l-.05,-.025"/>
<path d="M0,1.055 l-.025,-.05 M0,1.055 l.025,-.05"/>
</g>
<g transform="translate(20,425) scale(400,400)">
<text x="-.050" y=".05">(0,0)</text>
<text x=".28" y=".05">1/3</text>
<text x=".62" y=".05">2/3</text>
<text x=".98" y=".05">1</text>
</g>
</svg>
```

Figure 2.2: CSS and SVG to make Figure 2.1.

Chapter 3

LaTeX

LaTeX is a markup language. It is essentially an upgrade of Donald Knuth's landmark typesetting language called TeX [7]. In the late 1970s Knuth realized that printing technology had neared a point at which very sophisticated typesetting could be done by computers. He realized further that mathematical and technical typesetting had been entirely ignored by the primitive programs of the day. He undertook to create a typesetting language that would be able to produce book-quality output for all subjects, including mathematics and science.

The original "plain TeX" was very powerful, but in many ways difficult to use. Leslie Lamport [8] created a package of additional commands to simplify and extend the abilities of TeX. This package was called LaTeX, and has become the defacto standard for typesetting highly technical documents, particularly those containing mathematical notation. It is so prevalent that many publishers of mathematical papers provide their own class and style files [9],[4]. LaTeX subsumes and includes plain TeX. While it alters some plain TeX commands and behavior, in the end most plain TeX commands work in LaTeX. In this text we will use the terms TeX and LaTeX more or less interchangeably. This is undoubtedly improper, but we use elements of both, and often we prefer certain plain TeX commands over their LaTeX replacements.

The "tags" of the markup language are identified by starting with a backslash (\), i.e. every LaTeX command or variable starts with a backslash. As with other markup languages, every LaTeX document has a preamble section and a body section. Unlike most other markup languages with which we are familiar, there are many commands in LaTeX that do not require, or even possess, closing tags.

In any computer language, it is common to make one's first application a so-called "Hello, world!" program. Such a document in LaTeX follows.

```
\documentclass{article}
\% Any text following a percentage sign is ignored - a comment
\begin{document}
Hello, world!
```

```
This is a second paragraph.
\end{document}
```

The structure of a basic LaTeX document is evident. The file must begin by declaring the class of the document – the "article" class here. Other possibilities include "book", "report", and "letter". Following that are preamble commands, which pertain only to formatting and new command structures. The body of the document begins with the `\begin{document}` statement. Any text that is not preceded by a backslash actually appears on the formatted page.

Note that the blank line in the body of the document makes a new paragraph. It is very important to watch blank lines - starting a new paragraph can generate errors if it is done e.g. in the middle of an equation.

Note also that there are several special characters that cannot be used in an ordinary way. They include \, _, ^ , {, }, &, #, ~ , %, and $. If you actually want one of these characters to appear in your document, you must enter them as `\textbackslash`, `\_`, `\^~` , `\{`, `\}`, `\&`, `\#`, `\~~`, `\%` and `\$`, respectively. Some of these characters are only valid in math mode, or require some extra tricks to manage.

3.1 Starting a document

The very first line of a LaTeX file is one of the most important. It sets many properties that apply to the entire document. In the `\documentclass` statement, we choose the basic format settings for the document. The choices include: `article, book, letter, report,` and `beamer`. The `article` class is designed for academic papers and class reports. It uses the same margins for both even- and odd-number pages, sets no extra page headers, and does minimal titles. By contrast, the `book` class makes an entire page for the title and author information, uses different margins for even- and odd-numbered pages, and puts the section heading at the top of odd-numbered pages, while the chapter heading goes to the top of even-numbered pages. We will discuss the beamer class in some detail later, but it is used to make a "powerpoint" style presentation.

There are two other classes that are useful for mathematicians and scientists. The `amsart` and `amsbook` classes replace the `article` and `book` classes, respectively, and are mostly interchangeable with those. The chief difference is that the AMS classes load special AMS formatting packages automatically. When we want to use those packages in e.g. the ordinary `article` class, we must load them explicitly:

```
\usepackage{amsmath,amsfonts,amsthm,amssymb}
```

The `\documentclass` command takes many optional arguments. Probably the most important is a specification for the font magnification. Specifying

```
\documentclass[10pt]{article}
```

typesets an article using a 10 point default font. All other fonts are scaled accordingly. In other words, when we change that option to `12pt`, then the entire page is scaled up in size by a factor of 1.2. All of the headings, footnotes, and other fonts that are different from the default are scaled. The point is that that specification is not so much a designation of the font size for the paper as a magnification. The only choices are `10pt`, `11pt`, and `12pt`.

Another useful option for the documentclass command specifies whether to place equation numbers to the left or right. These options are `leqno` and `reqno`, respectively. Note that the ordinary `article` and `book` classes place equation numbers to the right by default. The AMS classes place them to the left.

TEX installations typically set default paper sizes according to the standards of the continent where they exist. European installations should set the default paper size to A4, which is somewhat longer and narrower than North American letter size. If you need to change the size of the paper that LATEX is to work with, you can set the `letterpaper` or `a4paper` optional arguments.

There are many other options, some of which are only valid for certain choices of document class. The `beamer` class in particular has a number of extraordinary options available. We will discuss these when we describe the beamer package.

After the `\documentclass` statement, there are often a number of lines of markup describing formatting options that apply to the entire document. This area of the file is often called the "preamble". After that, we must start typesetting. That is done by placing a `\begin{document}` statement where the actual typesetting should start. When we do that, we should insert an `\end{document}` line after it. Anything not between those two lines will never be seen when the typeset document is viewed.

Note that there are many default settings hidden in the `\begin{document}` command. For example, it sets line spacing, and some other dimensions. This might occasionally override the definitions made in the preamble to the document.

3.2 Commands and Environments

We have already noticed two different kinds of markup in a document. The `\documentclass` statement was a command, or in TEX, a *macro*. Macros are keywords preceded by a backslash. They can take arguments, which are typically surrounded by braces. Thus, typing `\documentclass{article}` alerts LATEX that a macro is coming when it sees the backslash. They keyword `documentclass` tells LATEX which macro it is dealing with. It can look that macro up and learns that it should look for one argument, which tells it which document class to use. We will use the words "macro" and "command" interchangeably.

Macros can often accept optional arguments, which are enclosed in brackets. Thus, typing `\documentclass{article}` is perfectly acceptable because LATEX expects exactly one required argument, but we can also add the optional arguments about magnification and paper size inside brackets: `\documentclass[11pt,letterpaper]{article}`. Again for emphasis: required arguments are enclosed in braces; optional arguments are enclosed in brackets.

The arguments to macros do not always have to be enclosed in brackets. If the argument is a single atom – a single character or TEX macro – then it need not be contained in braces. We shall see later that `\frac{1}{3}` and `\frac13` both produce $\frac{1}{3}$.

A second kind of markup that LATEX uses is the environment. Our example here is the `\begin{document}` statement. Environments are characterized by the use of the `\begin` macro. The `\begin` macro always puts some formatting into place that persists until an `\end` macro stops it. The formatting is specified by the name of the environment.

The begin statement for an environment can itself take required and optional arguments. For example, we shall see later that to make a table with two centered columns we start an environment `\begin{tabular}{cc}`; to make a floating region containing a figure we can use `\begin{figure}[ht]`.

3.3 Headings

Creating headings in LATEX is easy, and the system tries to help by keeping track of many things for you. For example, you could create headings as in simpleminded word processing programs by typing text directly and imposgin some formatting such as a larger font or centering, but LATEX prefers to memorize those headings, count the chapters and sections for you, and then offer those titles and numbers for the sake of reference whenever you want them. In short, LATEX is more aware of the structural details of a document than typical word processors.

In most document classes, the best way to create a title area for documents is to set the values for the text to appear there in the header. For example we could typeset the title for this document using these commands.

```
\title{Essentials of Scientific Computing}
\author{Kevin Cooper}
```

This does not typeset those values anywhere. In order for the title area actually to appear in the document, we must invoke `\maketitle` inside the document, i.e. between the `\begin{document}` and the `\end{document}` statements. Needless to say, most people would put this statement as the first thing in the body of the document.

Inside the document, we can create headers using commands such as `\chapter` or `\section`. These commands accomplish several things: they typeset the appropriate heading in some uniform way, according to the specifications of whatever class we are using; they number the heading; and they allow us to reference that number if we need to. The headings for which this works include `\part`, `\chapter`, `\section`, `\subsection`, `\subsubsection`, `\paragraph`, and `\subparagraph`. The `article` class does not include the `\chapter` heading.

Sometimes we want the heading, but we do not want it to be numbered, or for the associated counter to be incremented. When that is the case, we can change the appropriate command slightly by appending an asterisk (*). Thus

```
\section*{My Unnumbered Section}
```

creates the header without all the numbering.

3.4 Mathematical Typesetting

LaTeX was created for typesetting mathematics. There are many fine points to mathematical typesetting that are easy to overlook when we simply read math books. We discussed these in Section 2.5, but recall that in order to typeset mathematics, we require a special “math italic” font set, with different spacing from ordinary text, we need to be able to do complicated alignments both horizontally and vertically, and we use a huge collection of special symbols, diacritical marks, and notations.

3.4.1 Math Mode

In TeX, as in MathML, we need to change to a special “math mode” whenever we need to type any mathematical symbol. In a line this is done by surrounding our mathematical text with dollar signs, as in `$x$`, or alternatively using a begin and end symbol: `\(x\)`. The dollar signs come from plain TeX, while the `\(` and `\)` come from LaTeX. If we want to *display* the expression (make it centered, on its own line, with extra space around it, and a more expansive format), we must use a display math environment:

```
    \begin{displaymath}
    f(x) = x+2.
    \end{displaymath}
```

or

```
    $$f(x) = x+2.$$
```

or

```
    \[f(x) = x+2.\].
```

or

```
\begin{equation*}
f(x) = x+2.
\end{equation*}
```

All of these produce the same thing:

$$f(x) = x + 2.$$

We will henceforth use the double-dollar-sign formulation to start display math mode without equation numbers. This comes from plain TeX, while the other forms are part of LaTeX.

To have your equations numbered automatically as well as set in a display mode, use the equation environment.

```
\begin{equation}
f(x) = x+2.
\end{equation}
```

This gives the same displayed equation as above, but with an equation number:

$$f(x) = x + 2. \tag{3.1}$$

Note that this is almost the same as one of the formats we described earlier – it differs by an asterisk from `\begin{equation*}...\end{equation*}`. This is something found often in LaTeX. There are many environments for which you can add an asterisk to the name, which makes any numbering disappear. Another example is the `\section*` macro that we discussed earlier.

3.4.2 Math Notation

One of the most basic differences between mathematical typesetting and ordinary text is the prominent use of subscripts and superscripts in math. This is very easy to do in TeX math mode. A subscript is created by placing an underscore (_) character followed by the subscript object. Thus `$X_1$` produces X_1; `$a_i$` produces a_i. If there is more than one object in the subscript, we must use braces to group the objects so that they appear as one thing to the subscript operator; e.g. `$a_{i+1}$` produces a_{i+1}, whereas `$a_i+1$` produces $a_i + 1$.

Making superscripts is very similar, but uses a caret (^) instead of the underscore. Thus, `$x^2$` produces x^2; `$e^{-x}$` produces e^{-x}.

We see here the use of grouping in LaTeX to make several objects appear as one to certain commands. Most commands in TeX require a specific number of inputs. When those input arguments are single characters or even single symbols then there is no ambiguity. However, when those arguments are more complex constructions, we use braces { and } to enclose them as a group.

Another way mathematical notation differs from ordinary typesetting is the common use of Greek letters and other special symbols. These are usually easy to guess: their LaTeX names typically are just their names spelled out, preceded by a backslash. Upper-case Greek letters spell the names using a capital letter. For example, `\lambda` produces λ, while `\Lambda` gives Λ. Note in particular that `\int` produces an integral sign, `\sum` produces a summation symbol (do not use `\Sigma`!), and `\infty` makes an infinity symbol. Most of these symbols can only be produced in math mode – do not try them in ordinary text.

Mathematics involves other special constructions not found in ordinary text, such as fractions and square roots. LaTeX provides macros to make all of these. The `\frac` command takes two arguments: the numerator of a fraction first, the denominator second. Thus `$\frac{1}{2}$` produces $\frac{1}{2}$. The `\sqrt` command takes only one argument, which it puts under a square root symbol, viz. `$\sqrt{b^2-4ac}$` produces $\sqrt{b^2-4ac}$.

When function or operation names are spelled out, as with sines, cosines and other trigonometric functions, or as with the limit symbol, then it is traditional to typeset these in a normal font, so that they look like words, rather than e.g. $c \cdot o \cdot s$. Thus, when we set the expression $\cos x$, we type `$\cos x$`. Just for comparison, note that `$cos x$` produces $cos x$, which probably does not express the mathematics very well. Likewise, the LaTeX macros `\sin`, `\log`, `\exp`, and `\lim` are all defined, as well as many others.

There are many mathematical constructions that stack symbols. We can usually typeset these by using only the subscript and superscript tags. In some cases these are displayed differently depending on whether they use in-line or display mode. For example, consider a simple summation. The sum

$$\sum_{n=1}^\infty \frac{1}{n^2}$$

looks different in a line of text, viz $\sum_{n=1}^\infty \frac{1}{n^2}$. We typeset this the same way regardless of whether it is displayed or in-line. It looks like

```
$\sum_{n=1}^\infty \frac{1}{n^2}$
```

and if we want it to be displayed, we just put another dollar sign at the beginning and end. Another construction that has this behavior is the limit. An in-line limit looks like $\lim_{x\to0}\frac{\sin x}{x}=1$; in display mode this looks like the following

$$\lim_{x\to0}\frac{\sin x}{x}=1.$$

In both cases, we typeset it as

```
$\lim_{x\to0}\frac{\sin x}{x}=1$
```

where only the number of dollar signs changes.

Sometimes we need English words in lines of mathematics. These are again typed using an ordinary font rather than math italic, so that they are not confused with variables. We use the `\text` command to accomplish that.

Be aware that we have to put the spaces for text typesetting in ourselves. A backslash followed by a space inserts a normal stretchable space in a location. For example, to typeset the definition

$$p_n(x) = \frac{x^n}{n!}\ \text{for}\ n=1,2,3,\ldots$$

we can use

```
$$p_n(x) = \frac{x^n}{n!}\ \text{for}\ n=1,2,3,\ldots$$
```

Frequently we want to enclose large expressions or matrices in parentheses or braces. For this we use the `\left` and `\right` commands. These commands both take one argument: the parenthesis, brace, or other object that is to enclose some body of text. They are aware of their partners. In other words, when you use `\left`, you *must* use `\right`. For example, here is a set.

```
    $$S = \left\{ 1, 2, \left( \frac{2}{3} \right) \right\}$$,
```

which produces

$$S = \left\{ 1, 2, \left(\frac{2}{3} \right) \right\}.$$

The point is that `\left` tells TEX: "stretch the following symbol as much as is needed for the objects it encloses." Note that piecewise function definitions and certain other notations require the use of a brace or other delimiter on one side of a collection of mathematical expressions, with no matching brace on the other side. When that happens, use e.g. `\right.` to close the `\left`-`\right` construction. The period does not appear in your document - this closes the construction without making the delimiter.

Sometimes we want to emphasize that two letters are not symbols being multiplied, or we want to have one symbol overlap another. LATEX provides thin spaces and negative space for this. The negative thin space command is `\!`. Likewise we want just a little extra space between symbols, and LATEX provides a positive thin space for this: the macro is `\,`. Thin spaces are useful for subtle typesetting effects required, for example, in integrals:

```
$$\int_0^x \!f(t)\,dt.$$
```

yields

$$\int_0^x \!f(t)\,dt.$$

3.4.3 Alignment

One of the more problematic issues in typesetting mathematics involves alignment. Often we need to align the equals signs in a set of equations, or we must break a single equation into more that one line. There are many mathematical constructions, such as piecewise defined functions and matrices, that have alignment built into them. Ultimately, all alignments are based on tables, with columns of text that are forced left or right. The basic LaTeX environment for aligning equations is “eqnarray”. This is not the best way to align equations, but we’ll look at it as an example, and for completeness.

```
\begin{eqnarray}
f(x) &=& (x-1)(x+1)\\
\nonumber &=& x^2-1.
\end{eqnarray}
```

This produces the following display:

$$\begin{array}{rcl} f(x) & = & (x-1)(x+1) \qquad (3.2) \\ & = & x^2-1. \end{array}$$

We should note several things here

- Think of the eqnarray as a table with three columns. The ampersand characters (&) separate the columns.
- We make a new line with the double-backslash. Thus, we should not put a double-backslash on the last line. The expression above has two lines - if we ended the second with a double-backslash, it would have three lines.
- Every line gets an equation number. If we do not want a separate equation number for a line, we should put the command `\nonumber` on that line.

The amsmath package provides a number of environments to do alignments. These are more versatile than “eqnarray”, and they are superior in appearance. In particular the “align” environment provides a better way to align operators for a collection of equations.

```
\begin{align}
f(x) &= (x-1)(x+1)\\
\nonumber &= x^2-1.
\end{align}
```

This renders the same equations as above, but the output looks like this:

$$\begin{aligned} f(x) &= (x-1)(x+1) \qquad (3.3) \\ &= x^2-1. \end{aligned}$$

- There is a little bit less space around the equals sign.
- This time the alignment only has two columns. The ampersand character (&) separates the columns.

- The align environment allows an arbitrary number of columns, so you can align more than one set of equations. Odd-numbered columns are right justified, while even-numbered columns are left justified. In another way of looking at it, odd-numbered columns represent the left side of equations, while even-numbered columns represent the right side of equations.
- Every line gets an equation number. If we do not want any equation numbers for a line, we should use the "align*" environment. That suppresses all equation numbers. The `\nonumber` macro works here as well.

Just to illustrate this more fully, let's do a multi-column alignment. We can typeset the following text from a numerical analysis book [5] using the align* environment.

$$\begin{aligned}
k&=0& x^{(0)}&=(&3&.000000, &7&.000000, &-13&.000000\,)\\
k&=1& x^{(1)}&=(&-0&.801653,&-0&.008264, &-1&.000000\,) &r_0&=-5.889\\
k&=2& x^{(2)}&=(&-0&.950887,&-0&.017735, &-1&.000000\,) &r_1&=1.19759
\end{aligned}$$

The LaTeX looks like this.

```
\begin{align*}
k&=0& x^{(0)}&=(&3&.000000, &7&.000000, &-13&.000000\,)\\
k&=1& x^{(1)}&=(&-0&.801653,&-0&.008264, &-1&.000000\,)
  &r_0&=-5.889\\
k&=2& x^{(2)}&=(&-0&.950887,&-0&.017735, &-1&.000000\,)
  &r_1&=1.19759\\
\end{align*}
```

The first ampersand pushes the k and 0 close to the equals sign. The second ampersand signifies that a new equation is starting – it puts in a larger space. The third ampersand pushes things close to the equals sign. The next ampersand says a new equation is beginning, even though it is not. As a result, it puts more space in. There is no easy way to avoid this, but it looks good enough. The alignment continues like that: odd-numbered ampersands squash the things around them close; even-numbered ampersands put some extra space in on the assumption that a new equation is coming. Thus, we align all the equals signs and all the decimal points. Note that we did put a thin space in at the end of the numbers in the parentheses to enhance readability.

The "align" environment starts math mode. Sometimes we need to do a complex alignment in a context that does not involve a matrix, and in which math mode must be started before the alignment occurs. In this case, the "aligned" environment allows us to do the formatting of the "align" environment, but in a math mode that is already in progress. For example, a book concerning linear algebra [11] has an equation that looks like the following:

```
$$
\left.
\begin{aligned}
&(A-\lambda_1I)^3x_1=0,&&(A-\lambda_1I)^2x_4=0,
  &&(A-\lambda_2I)^2x_6=0,&&(A-\lambda_3I)x_8=0\\
&(A-\lambda_1I)^2x_2=0,&&(A-\lambda_1I)x_5=0,
  &&(A-\lambda_2I)x_7=0\\
&(A-\lambda_1I)x_3=0
\end{aligned}
\right\}.
$$
```

This produces

$$
\left.
\begin{aligned}
&(A-\lambda_1I)^3x_1=0,&&(A-\lambda_1I)^2x_4=0,&&(A-\lambda_2I)^2x_6=0,&&(A-\lambda_3I)x_8=0\\
&(A-\lambda_1I)^2x_2=0,&&(A-\lambda_1I)x_5=0,&&(A-\lambda_2I)x_7=0\\
&(A-\lambda_1I)x_3=0
\end{aligned}
\right\}.
$$

Note that in this case we are aligning the left edges of equations. Since we want every column of equations to be left justified, we require text to appear only in even-numbered columns. The double-ampersands effectively skip odd-numbered columns.

A matrix is another kind of alignment. There are several ways to do this. The LaTeX way is to use the "array" environment. This does not put in parentheses or brackets - they must be inserted using `\left` and `\right`. Alternatively, if using the amsmath package, there are "pmatrix" and "bmatrix" environments, the first for a matrix encompassed by parentheses, the second for one encompassed by brackets. The columns of the matrix are again separated by ampersands; rows of the matrix are ended by double backslashes. The code

```
    $$
    A=
    \begin{pmatrix}
    1 & 2\\
    0 & 4
    \end{pmatrix}
    $$
```

produces the matrix

$$
A=
\begin{pmatrix}
1 & 2\\
0 & 4
\end{pmatrix}.
$$

Another kind of alignment that arises often is for a piecewise defined function. This could be handled relatively easily using the "aligned" environment, or we could use the "array" environment, but the amsmath package provides an even easier way: the "cases" environment. For example, we could define the unit step function using the syntax

```
$$
u(x)=
\begin{cases}
0 & x<0\\
1 & x\ge0.
\end{cases}
$$
```

This produces the result

$$u(x) = \begin{cases} 0 & x < 0 \\ 1 & x \geq 0. \end{cases}$$

3.5 Font modifications

We have already noted that the documentclass statement allows us to specify the size of the default font. This constitutes so-called normalsize. To change the size of fonts inside the document, you must group the text you want to change, and then use a new font size specifier. For example, we can make tiny text. When we type `{\tiny text}` we get text. The size specifiers are: `\tiny`, `\scriptsize`, `\footnotesize`, `\small`, `\normalsize`, `\large`, `\Large`, `\LARGE`, `\huge`, `\Huge`.

Changing font styles and weights is similar. For example typing `{\it italic}` produces *italic* text. The styles include `\it`, `\bf` for bold face, `\tt` for typewriter monospace, and `\sc` for a small-caps style. Thus, in TEX typing `{\sc Small Caps}` produces SMALL CAPS. That is a plain TEX way of doing things. In LATEX, one typically uses macros for this: `\textit`, `\textbf`, `\texttt`, `\textsc`. Thus, `\textsc{Small Caps}` again produces SMALL CAPS.

If you want to change the default fonts throughout a document, there are a couple of ways to do that. To do this, we need to know a very little bit about the way LATEX classifies fonts.

LATEX has three basic font classifications: roman, by which TEX means serif; sans-serif, and teletype, by which TEX means monospace. We can set the default fonts for each of these classifications respectively using e.g. the commands:

```
\renewcommand{\sfdefault}{phv}
\renewcommand{\rmdefault}{ptm}
\renewcommand{\ttdefault}{pcr}
```

These commands tell LATEX to use a `phv` (a Helvetica) font family for all sans-serif fonts; to use `ptm` (a Times)family for all serif fonts; and to use `pcr` (a Courier) for all monospace fonts. Unfortunately, to use these commands directly requires us to know exactly which font substitutions we want. Even finding this information can sometimes be difficult.

Probably the easiest way to change fonts throughout a document is simply to load a different font package. Somewhere in the preamble you can type the command e.g.

```
\usepackage{times}
```

that includes the name of a font family that is supported. The collection of these packages will vary from one TEX installation to the next. In the very common "texlive" distribution, the families available include avant, charter, pifont, bookman, courier, newcent, times, chancery, helvet, palatino, and utopia. A few of these might not show up properly when you first request their packages. This is often because they do not specify fonts in all three classifications. That might mean that you have to change your choice of default font family. For example, simply loading the helvet package does not automatically change your document default font to Helvetica – you must also tell LATEX that you want to use a sans-serif as your default font family. You can do this using the commands

```
\usepackage{helvet}
\renewcommand{\familydefault}{\sfdefault}
```

3.6 Counting

As we saw in Section 3.4, LATEX counts many things, such as including chapters, sections and other document parts, equations, citations, theorems, and so on. We can manipulate these counters using the `\setcounter` and `\addtocounter` macros. This is important for several reasons. First, it seems obvious that we do not want to have to assign the equation numbers ourselves. If we were forced to number section, theorems, and equations ourselves, we would constantly have to renumber everything as we edited the file. Insert an equation, and then change all the numbers. Change your mind and delete that equation – change all the numbers back. It is very helpful for TEX to do that for us. Second, it is difficult or impossible to remember the numbers of all the equations. It is easier to remember names. Finally, when we refer to equations and theorems by name, then when we insert that new equation, TEX not only renumbers for us, but changes the references. This is a huge time saver, and can prevent embarrassing errors.

TEX always registers the name of whatever counter applies to the current element. We can assign a label to the value of that counter using the `\label` macro. We can then get that value by using the `\ref` macro. To label a counter, all we need to do is to put the `\label` macro in the area where that counter applies. It is safest to do this right after the counter is incremented. Thus we want to label a section right after it starts, we label equations right after the `\begin{equation}`, and so on.

For example, in this section the counter that applies is the section counter, which has the value 3.6.

We place the macro `\label{sec:counting}` in a line after the `\section`, and then refer to it using `\ref{sec:counting}` Note that it is not necessary to put the `\label` command immediately after the `\section` command. There are many places throughout the section where you could put it. However, it is safe to put it right at the beginning. The name of the label is just something we made up. We could have called it "hubert" if we wanted to, however using the name "sec:counting" gives us a hint that this is a section number, and it is the section about counting. The point is that in a long paper or a book like this, there are hundreds of labels, so it is important to name them well.

In order to refer to values that have been assigned to labels, you should run LaTeX twice: once to get the value into the .aux file, and then again to get it back out for the reference. Many TeX editors do this for you – for example, Texstudio and Texmaker just compile until all the equations are numbered properly.

This works just as well for equations. If we set a `\label` inside a numbered equation, that means the equation counter is the most recently updated, so that the equation number will be assigned to the label. For example, there is a label on equation (3.3). Be sure to put the label on the line that is numbered.

We can always create counters of our own using the `\newcounter` command. This simply defines a counter – to initialize it, use `\setcounter`. From then on, we may increment it using `\stepcounter`. For example, we could define a counter called counttopic, and set its value to 24 using the commands

```
  \newcounter{counttopic}
  \setcounter{counttopic}{23}
  \stepcounter{counttopic}
```

Since the counter is probably not associated with a text element, to get its value we probably need to use the `\arabic` macro. This evaluates the counter and typesets the resulting number using arabic numerals. There are corresponding macros called e.g. `\roman` and `\Roman` to typeset the counter using lower or upper case roman numerals. The current arabic value of the counttopic counter is 24, while in upper case roman numerals it is XXIV. If you prefer to represent the counter using letters, you could use the `\alph` or `\Alph` macros, which currently have values x and X respectively.

There are several environments that produce other counters. In particular, the theorem environment is essential for writing rigorous mathematical papers. In general we can create a structure for numbering theorems and such using the `\newtheorem` macro. In this section we have typed

```
\newtheorem{Thm}{Consequence}[section]
\newtheorem{Def}[Thm]{Fact}
```

This creates two new theorem environments. The first is associated with a name `Thm`, and every instance of it will start with the word "Consequence" typeset in a bold face. The optional `section` argument here indicates that the numbering for the `Thm` counter will appear after chapter and section numbers, with period separators. Likewise, the next line defines another theorem environment whose name is `Def`. Every instance of this will start with the bold face word "Fact". The optional `Thm` argument in this position indicates that this new environment will share the `Thm` counter.

Here is an example.

Fact 3.6.1 *Any offset text in italic style, with a number by which to refer to it, is treated as a theorem by LaTeX.*

Consequence 3.6.2 *Definitions, propositions, theorems, corollaries, lemmas, and many other textual elements can all be treated using theorem environments.*

These are typeset as follows.

```
\begin{Def}
Any offset text in italic style, with a number by
which to refer to it, is treated as a theorem by \LaTeX.
\label{thm:description}
\end{Def}
\begin{Thm}
Definitions, propositions, theorems, corollaries,
lemmas, and many other textual elements can all
be treated using theorem environments.
\label{thm:justification}
\end{Thm}
```

With that we can refer to Fact 3.6.1 using `\ref{thm:description}`, and to Consequence 3.6.2 using `\ref{thm:justification}`.

3.7 Lists

In most markup languages, there are essentially three kinds of lists: ordered, unordered, and description. In LaTeX, these are created using environments called `enumerate`, `itemize`, and `description`, respectively.

An ordered (enumerate) list looks like the following:

1. Enumerate
2. Itemize
3. Description;

while an unordered (itemize) list looks more like:

- Enumerate
- Itemize
- Description.

In both cases, individual items in the list are preceded by the `\item` macro. The numbering happens automatically. Thus, to typeset the ordered list, we use

```
\begin{enumerate}
\item Enumerate
\item Itemize
\item Description;
\end{enumerate}
```

We can nest these lists at will. LATEX keeps track of all the numbering for us, and even changes the way counters are displayed according to the level of the list. For example, the nested list

```
\begin{enumerate}
\item Enumerate
  \begin{enumerate}
  \item Numbered items
  \item Lettered items
\end{enumerate}
\item Itemize
\begin{itemize}
  \item Bulleted items
  \item Boxed items
\end{itemize}
\item Description
\end{enumerate}
```

displays as

1. Enumerate

    (a) Numbered items

    (b) Lettered items

2. Itemize

    - Bulleted items
    - Boxed items

3. Description

We can gain a greater measure of control over ordered lists by loading the enumerate package:

```
\usepackage{enumerate}
```

This allows us to insert an optional argument for the enumerate environment to act as a template for the numbering format we want. For example

```
\begin{enumerate}[a)]
\item Numbered items
\item Roman numeral items
\item Letter items
\end{enumerate}
```

produces

a) Numbered items

b) Roman numeral items

c) Letter items

while changing the first line of the list code to

```
\begin{enumerate}[I:]
```

produces

I: Numbered items

II: Roman numeral items

III: Letter items

We can see the simplicity of the package. The optional argument will order the list by letters of the alphabet if it contains the letter "a". If the "A" is capitalized, then the ordering is by upper case letters; otherwise lower case is used. If instead the optional argument contains "i" or "I", then the list is ordered by lower case or upper case Roman numerals, respectively. Naturally, a "1" in the optional argument produces an ordinary arabic enumeration. Any other characters found in the optional argument appear verbatim in the list.

We see that the last feature provides an easy way to change the "bullets" on an unordered list as well. For example, if we wanted to use a math `\clubsuit` symbol instead of the usual bullets, we could change the first line of the environment to

```
\begin{enumerate}[$\clubsuit$]
```

Note that we must still use the enumerate environment, but the numbers do not show up now. Instead, we see

♣ Numbered items

♣ Roman numeral items

♣ Letter items

There are more powerful and sophisticated packages available for changing the enumeration or bullet style of a list, but these are also somewhat more complicated to use. Look at e.g. the enumitem package if you want something fancier.

The description list uses a slightly different syntax. Each individual item still is preceded by the `\item` macro, but this time it accepts an optional argument giving the term to be described. Thus, the first line below is `\item[First type:] Enumerate`.

First type: Enumerate

Second type: Itemize

Third type: Description

3.8 Boxes, Glue, and Space

It is probably time to discuss the way that LaTeX assembles text into lines and pages, and learn something about how we can change that. This is a somewhat obscure topic, but it is necessary to understand this if we are to make sense of how TeX places things on pages.

TeX fills a page using elementary building blocks, patiently assembled to make larger blocks called words, lines, and then fitting those together those into pages. The smallest building blocks are obviously individual characters and letters. Each character is associated with a box that has certain dimensions. It rests on the *baseline*, and has some height above that line, and a depth below it. It has a width, which is fixed and uniform for monospace fonts such as Courier, but can vary from character to character otherwise. This is illustrated in Figure 3.1. These small letter boxes are pasted together by TeX into word boxes using stretchable space, called *glue*. The words, in turn, are assembled using more glue into lines. The point of allowing the space to stretch is to allow TeX to bump the text up against both the right and left sides of the page – to *justify* the text. The amount of glue TeX uses to assemble a line varies according to the total width of the boxes containing the letters and words. Figure 3.2 illustrates this process.

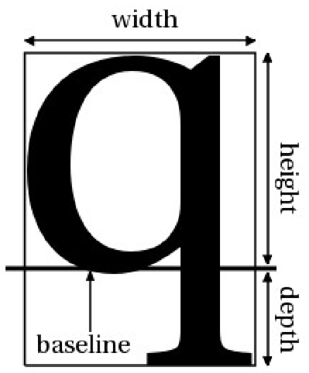


Figure 3.1: A box encompassing a character

Throughout the assembly of a line of text, LaTeX is working only horizontally. Once it has a couple of lines manufactured, it then must

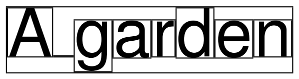


Figure 3.2: Assembly of letter boxes into lines

fit them together vertically. This is straightforward, but it can be important for us to understand the switch between horizontal and vertical mode. LaTeX always finishes its horizontal mode work before it is willing to think about anything having to do with vertical spacing.

The simplest way for us to change spacing is to insert one stretchable space. For this we use the `\ ` (backslash-space) macro. The space involved does not stretch much, but it can change size slightly to fill an alloted area. If we need to make a fixed-width, unbreakable space, we use instead the ˜ character. Remember that LaTeX ordinarily pays no attention to spaces except insofar as they delimit words and other text objects. TeX always thinks that it is the authority on how best to construct a page of text.

If we need to put a longer horizontal space into a line, we can use a `\quad` or `\qquad`. These names are old printers' terms. A quad is the width of the current font size - i.e. if the font size is 12pt, then a quad is 12 points. The name is probably a reflection that this is the length of a side of a square in the current font. This is not coincidentally the same as one em – the width of the a capital M. Thus, typing `\quad` produces one quad (one em) of space. A qquad is two quads. If we need different horizontal spaces than this, we can make these manually using the `\hspace{dimen}` command. The `dimen` parameter represents some distance. Thus we can put a one centimeter space between the words "centimeter" and "space" by typing `\hspace{1cm}` between them.

Vertical space is almost as easy to come by. There are several commands to create small vertical spaces: `\smallskip`, `\medskip`, and `\bigskip`. The amounts of space these provide varies with the document class. If you need a different amount, you can use `\vspace`. Thus, to put an extra 1 centimeter of space after this paragraph, we can type `\vspace{1cm}` somewhere near the end of it. Note that vertical space is never inserted until TeX is in vertical mode, i.e. when it gets to the end of a line. Thus, even if we type our `\vspace` command a few words before the end of the paragraph, the space will appear after the line the command is in finishes.

Note that LaTeX is quite happy to work with negative space. This can be used to put text in a margin, or to overlap symbols. For example, one way to typeset the word "two" is to type

```
\hspace{2em}o\hspace{-1.3em}w\hspace{-1.1em}t\hspace{1em}.
```

We do not recommend this method, but it illustrates the possibilities.

Since TeX works on boxes from the level of individual letters all the way up to vertical mode lines and paragraphs, it would surprise us if it did not provide a way to create boxes of arbitrary size. Naturally it does. the `\makebox` command creates a horizontal mode box of any width we specify. The `\makebox` command accepts two optional parameters: a width, and a position. The obligatory argument is the text to appear in the box. The position can be “c” for centering the text in the box, “left” to left-justify the text, “r” to right-justify it, or “s” to spread the text throughout the box. Here are some examples.

```
\makebox{a fox in a box}
\makebox[2in][r]{a fox in a box}
\makebox[2in][l]{a fox in a box}
\makebox[2in][s]{a fox in a box}
\makebox[0.4in][s]{a fox in a box}
```

This produces the lines illustrated in Figure 3.3. Notice that when the width of the box is set to be less than the width of the text, then the text is simply allowed to overlap the box.

There is another macro called `\framebox` that does much the same thing as `\makebox`. The only difference is that it puts a border around the box. If you want to display text or other single-line content in a frame, use the `\framebox` macro.

It is important to note that both `\makebox` and `\framebox` operate only in horizontal mode – they can only typeset one line. Occasionally we want to put an entire page area in a box of a specified size, complete with line breaks and justification. LaTeX provides two principal ways of doing that. The `\parbox` macro takes two arguments: the width of the “paragraph box” to be created, and the content of that box. The minipage environment is another way to create such areas.

```
a fox in a box
                 a fox in a box
a fox in a box
a     fox     in     a     box
a fox in a box
```

Figure 3.3: The results of various `\makebox commands`

The idea for the minipage is simple enough. It is an environment that takes one obligatory argument, which contains the width of the area. The environment itself contains the content to be typeset. In Figure 3.4 the minipage on the left shows the LaTeX code to display the minipage on the right. Since each minipage has a width of less than half the width of the page, they appear side by side. The point is that a minipage is treated as just another horizontal mode element to be lined up with others.

This last point is worth emphasizing. The minipage (or parbox) grouped a lot of content into a single box. That box might be fairly tall – indeed the point is to make a box that contains more than one line. On the other hand, that box is treated as a single element for LaTeX to assemble into a line in horizontal mode. Thus, we could put a frame around the minipage using a `\framebox`. Again, the `\framebox` command can only work in horizontal mode, but since the minipage has grouped vertical material into a single element that is typeset in horizontal mode, the `\framebox` accepts it. There are many other horizontal mode tricks that can be applied once you have concentrated your vertical problem into a minipage.

```
\begin{minipage}
{.45\textwidth}
*\hfill**\\
*\hfill*\hfill*\\
\vfill
*\hfill*\hfil*
\end{minipage}
```

```
*                         **
*            *             *





*                         **
```

Figure 3.4: Two minipages used to illustrate `\hfill` and `\vfill`

Often in arranging minipages and other content on a page, we need to separate them by "as much space as is available." We could just guess the amount of space for an `\hspace` command, look at the result in LaTeX, then change the amount until the result looks right. That would be crazy. It would take too much time and trouble, and the first time we changed anything we would have to start all over with the process. Fortunately, TeX provides several commands that make "as much space as is available." These are `\hfill`, `\hfil`, `\vfill`, and `\vfil`. The commands whose names contain two "l"s are in some sense stronger than those with only one.

The use of these commands is illustrated in Figure 3.4. We see that the `\hfill` command simply separates the characters to its left and right by as much space as it can fit on a line. Note that if there is no character to its right, then it does nothing – putting an `\hfill` at the end of a line would be pointless. The `\hfil` command does the same thing, with a couple of special rules.

- `\hfill` can make space up to the entire width of the text; `\hfil` can only go to half the width of the text.
- If n `\hfill` commands are found on the same line, they each take $1/n$ of the available space.
- If `\hfill` and `\hfil` are found on the same line, `\hfill` takes all the space, and `\hfil` gets none.

The `\vfill` and `\vfil` commands work on the same principles. Remember that these are vertical mode commands, so they only take effect after horizontal mode is finished, i.e. after a line has been completed.

There are other commands that work in a similar way. The `\hrulefill` command makes a horizontal rule that expands to fill the space between two characters. The `\dotfill` command does the same with dots. Note finally that the `\rule` command also makes horizontal rules, but this command takes width and height arguments. Thus,

```
\rule{.33\textwidth}{0.5pt}
\makebox[.33\textwidth]{\hrulefill}
```

makes two identical horizontal rules, each 1/3 of the page in width.

3.9 Page Layout

In Section 3.8 we used a TeX object called `\textwidth` in several places without explanation. At the time, we took it to be largely self-explanatory, but it is time to discuss it now. It turns out that this is not a TeX command; instead it is a variable that has a certain value. Initially that value is set by the document class, but we can change it if we like. It is just one of the variables that control the page layout in LaTeX.

Page layout in TeX is not quite as transparent as in wysiwyg word processors. The first thing to note is that TeX does not know how big the page is. It only knows where the text area starts, and how big it is. For this purpose it maintains several variables whose values can be manipulated. They are

- oddsidemargin – the left margin on odd-numbered pages, measured from a normal of 1 inch. In the `article` document class this is the only margin variable used.
- evensidemargin – gives the left margin on even numbered pages, from a normal of 1 inch. This applies only in certain document classes, e.g. the book class.
- topmargin – the top margin, from a normal of 1 inch. Note that there is also a header area below the top margin.
- headheight – the height of the header.
- headsep – the distance by which the header is separated from the text area.
- textwidth – the width of the text area.
- textheight – the height of the text area.
- footskip – the distance skipped to the footer.
- baselineskip – the distance from one baseline to another. Note that this variable is typically set in the `\begin{document}` statement, so if we want to change it, we usually cannot do that in the preamble.
- baselinestretch – stretches the baselineskip. 1.0 would be the default, 2.0 would give double spacing. This must be set using `\renewcommand`, since it is not a dimension. This is considered to be the preferred way to change the line spacing in LaTeX.
- parskip – the distance between paragraphs.
- parindent – the amount of indentation in the first line of a paragraph.

We can set values in these variables in several ways. Even in plain TEX it was possible to assign values directly: e.g. `\textwidth=6.5in`. This works in LATEX as well. In LATEX we have other options. We could replace the equals sign with a space: `\textwidth 6.5in`. Or again, we could use a `\setlength` command: `\setlength{\textwidth}{6.5in}`.

3.10 Figures

LATEX does not naturally understand anything about graphics. Before we can draw figures or display images, we must load a package that can do that. There are a few packages that handle aspects of the task, but the one almost universally used now is called graphicx. Thus, in the preamble of our document, we type

```
\usepackage{graphicx}
```

Once the graphicx package is loaded, it provides a useful macro called `\includegraphics` that loads any image file into the document. There are, of course, a few provisos.

- The `pdflatex` program only admits image files. In particular, it cannot use encapsulated postscript.
- The ordinary `latex` program cannot use any file that does not print directly; i.e. it cannot use image files – only encapsulated postscript.
- The filename parameter contains a path to the file we want relative to the directory this document resides in. The easiest way to do this is to have the image file in the same directory as the document, and then just type the file name.
- The file name should not be enclosed in quotes.

Thus, typing

```
\includegraphics{filename}
```

at any point in the document inserts the image contained in `filename` at that point. The image is inserted in horizontal mode, so you might want to put it in its own paragraph, or at least on its own line.

The `\includegraphics` command accepts an optional argument, where we can specify scaling, width, height, or other attributes. It might be that the most practical way to handle the size of the image is to decide how much horizontal space you have for its display, and specify the height accordingly. For example, we might decide that we are willing to allow our image to take up to 50% of the width of the page. In that case we would type

```
\includegraphics[width=.5\textwidth]{filename}
```

Figure 3.5: A scene from the Wind River Range

This tells TeX to make the image 1/2 of the value of `\textwidth` wide.

Unfortunately, just sticking an image into the document at any old spot is probably not a good plan. It might not fit well, resulting in a large blank space at the bottom of a page, or having the image encroach on the margins. Perhaps we would play around with it until we could make it fit well, but that would take significant time. We might want a caption for the image, which would then need to be aligned in some way. Again, it is traditional in academic papers to number the image and refer to it by number – we should not refer to "the image below." LaTeX handles all of these issues by providing floating areas in which we can put images, and building captions with counters into those. The most-used such utility is the figure environment.

The figure environment does not have much to do with figures. It simply creates a floating region that is as wide as the current `\textwidth`. It does permit a `\caption` command that increments the `figure` counter and inserts a caption. Once we have the floating region created, we can insert anything we want into it. Figure 3.5 was inserted using the following code.

```
\begin{figure}[t]
\begin{center}
\includegraphics[width=.6\textwidth]{windrivers.jpg}
\caption{\label{fig:winds} A scene from the Wind River Range}
\end{center}
\end{figure}
```

The optional parameter `t` following the the figure environment declaration tells TeX that we want the figure to appear at the top of a page. Note that it might not be the same page where the text above or below falls, but remember, we can refer to the figure using the label in the caption command. It is somewhat important to put the label *inside* the caption – remember that the figure counter is not incremented until we invoke the caption command.

This is not the only way to put a figure into our document. Often we want to place a figure to one side, and have the text flow around it, as we did with HTML. This too is possible if we first load the wrapfig package:

Figure 3.6: A view of the Cascade Range

```
\usepackage{wrapfig}
```

This package defines the wrapfigure environment. Observe the difference in names: the package name is wrapfig, but the environment is wrapfigure. The use of this environment differs slightly from the plain figure. Here we must tell the environment which side of the page to use to display the image, and how wide the area should be. Thus there are two obligatory arguments to the wrapfigure environment: a letter indicating the position first, then the width of the area. There are several choices for the positioning letter: r, l, i, or o. Obviously "r" stands for right; "l" for left. However, sometimes we want our picture to be on the inside (near the book binding) of a page. That is what "i" specifies. Then "o" tells LaTeX to put the area on the outside of the page. These lower case letters all indicate that the image upper left corner should start exactly where the code appears in the text; using upper case letters would allow the wrapfigure region to float.

The wrapfigure environment often leaves too much vertical space above and below its content. This can be dealt with clumsily but easily by inserting a few `\vspace` commands with negative arguments. The code for inserting Figure 3.6 was as follows.

```
\begin{wrapfigure}{I}{.45\textwidth}
\begin{center}
\includegraphics[width=.40\textwidth]{easypass.jpg}
\caption{\label{fig:easypass} A view of the Cascade Range}
\vspace{-1cm}
\end{center}
\end{wrapfigure}
```

Recall that the `\includegraphics` command inserts images in horizontal mode. Thus, aligning images side by side is quite easy. However, assigning and referring to captions for those individual images can be more problematic. There are several packages that help with this, but most of them are clumsy and fragile. The subcaption package is probably the best of these, though it is not included in some popular TeX distributions. You might have to install it separately. The truth is that it might be easiest to handle all this manually, and forget the numbering for individual figures.

The LaTeX code to produce Figure 3.7 (a) and (b) follows.

Figure 3.7: (a) The Wallowa Range (b) The Cascade Range

```
\begin{figure}[h]
\includegraphics[width=.45\textwidth]{wallowas.jpg}
\hfill
\includegraphics[width=.45\textwidth]{cascades.jpg}
\caption{(a) The Wallowa Range\hfill (b) The Cascade Range}
\label{fig:wallowa+cascade}
\end{figure}
```

3.11 Tables

When TEX first appeared, one of the most difficult things to do with it was to make a table. This became one of the most important things driving people to adopt the LATEX package when it appeared – it made table creation much easier. Nonetheless, tables remain something of a sore spot for TEX. There are many packages that strive to improve their appearance and ease of creation, but it is still more troublesome to make good tables in LATEX than in word-processing programs or even HTML.

It is important to understand at the start that the table environment does not produce a table. Instead it is like the figure environment: it creates a floating region in which one can put a table. Indeed, the only differences between the table and figure environments is that they use different counters, and one counter is labeled "Table" while the other is not. The table environment accepts the same optional arguments as the figure environment.

The environment that we use to create actual tables is called "tabular". This environment takes an obligatory argument that describes the columns of the table and the separators around them. This argument understands only a few specifiers, which are listed in Table 3.1. The idea is simply to place vertical bars wherever the table will have a line separating columns or acting as a border, and then to place l, r, or c characters according to whether columns of text should be left or right justified, or centered.

By default, LATEX does not allow table cells to occupy more than one line.

Table 3.1: Table column specifiers

Symbol	Significance
|	a vertical line, used at the edge of the table or as a column separator.
l	left-aligned column
c	centered column
r	right-aligned column
p{w}	a column containing paragraph text with width w, aligned at the top
m{w}	a column containing paragraph text with width w, aligned in the middle. This requires the array package.
b{w}	a column containing paragraph text with width w, aligned at the bottom. This requires the array package.

However, if we need cells that contain multiple lines then LaTeX allows one format specification that does that. The `p{w}` format specifier makes a paragraph in a single cell. The text is vertically aligned at the top of the cell. The argument `w` is the width of that column. If we need the text to be aligned differently, we can load the array package to enable two other macros: `m{w}` aligns the text vertically in the middle of the cell, while `b{w}` aligns it at the bottom.

Table 3.1 lists these format specifiers. The code to generate that table follows.

```
\begin{table}
\begin{center}
\begin{tabular}{||l|p{.5\textwidth}||}
  \hline
  Symbol & Significance\\
  \hline
  $\vert$ & a vertical line, used at the edge of the table or
  as a column separator.\\
  l & left-aligned column\\
  c & centered column\\
  r & right-aligned column\\
  p\{w\} & a column containing paragraph text
  with width w, aligned at the top\\
  m\{w\} & a column containing paragraph text
  with width w, aligned in the middle.  This
  requires the array package.\\
  b\{w\} & a column containing paragraph text
  with width w, aligned at the bottom.  This
  requires the array package.\\
```

Table 3.2: Study of cold-stressed salmon

	Weights (gm)	
Day of Study	Control	Low Temp.
36	4.2	4.3
0	8.5	4.7
4	8.6	4.9
16	9.4	5.6
26	9.8	6.0
49	10.3	7.1
80	11.2	7.5
208	15.5	12.5

```
  \hline
\end{tabular}
\caption{\label{tab:tablespec} Table column specifiers}
\end{center}
\end{table}
```

Note that we must put the horizontal lines in ourselves using the `\hline` command. As with all of our alignments, the character that indicates column separation is the ampersand (&), and we end lines with the double-backslash macro.

Sometimes we need to have one table cell that stretches across more than one column of the table. LaTeX provides the `\multicolumn` command for that. This takes three arguments: namely the number of columns spanned by this cell; the format for this collection of cells; and the content for the cells. Likewise, sometimes we need to have a horizontal line that does not cross all the columns. For this we can use the `\cline` command. The argument here comprises the starting and ending columns, separated by a hyphen. This is illustrated in Table 3.2. The most important code to generate that table follows.

```
\begin{tabular}{|l|c|c|}
\hline
&\multicolumn{2}{c|}{Weights (gm)}\\
\cline{2-3}
Day of Study & Control & Low Temp.\\
\hline
36 & 4.2 & 4.3\\
0 & 8.5 & 4.7\\
      ...
208 & 15.5 & 12.5\\
\hline
\end{tabular}
```

The ability of LaTeX to create tables is tremendously more powerful than we have described here, however, most of that power comes from a large collection of extra packages that have been developed over a long period of time. In particular, the array package adds a number of useful capabilities, some of which we have discussed in Table 3.1. The multirow package adds the ability to make a cell in one column than spans many rows of the table. The tabularx package redefines and inserts commands to give a wider variety of formatting specifications, and to predefine certain aspects of the format. The xcolor package allows us to specify the color of individual rows or cells of tables. All of these packages are rather specialized, and we will not discuss them further.

3.12 Programming

One of the most powerful aspects of LaTeX lies in our ability to create new macros for specialized tasks. We can save ourselves typing and make complicated environments. While, as usual, there are many packages to enhance this capability, the basic tools are simple: they are the `\newcommand` and `\newenvironment` macros.

One of the simplest uses of `\newcommand` is to save typing. In the same way as we write "LaTeX" using a macro so as to avoid all the extra typing of the different commands required to make that, we can define new commands with short names to avoid typing long, commonly used expressions. For example, in a paper about ponderosa pines, one might would presumably need to type the scientific name of the species, *P. ponderosa*, very often. In LaTeX this is typed as `\textit{p. ponderosa}`. We could instead make a new macro called `\pp` that would typeset that for us.

```
\newcommand{\pp}{\textit{P. ponderosa}}
```

Thereafter, any time we need to refer to the scientific name, we would just type `\pp`. Note that if there is a space after the scientific name, we must put it in using the backslash-space command. This is so that when *P. ponderosa* is followed by a period or comma, those punctuation marks will be right next to the text.

Often we need to use some complicated typesetting commands that apply to different bits of text. If we want e.g. to automate the typesetting commands and apply them to the text we put in, then we must design a macro that accepts arguments. For example, in this text we have needed to typeset HTML commands. This requires us to use a typewriter font, and to use less-than and greater-than signs that TeX cannot typeset as we type them. We define a macro

```
\newcommand{\ht}[1]{\texttt{\textless #1\textgreater}}
```

This tells `\newcommand` that our new macro requires one argument, the value of which is inserted at the point where the #1 appears in the definition. In that case, `\ht{ul}` produces the text `<ul>`. We can use as many arguments as we require, and the definition can extend over multiple lines. As another example, in this text we constantly need to type LaTeX commands. We do so by using a specialized macro called `\tc`. Its definitiion is

```
\newcommand{\tc}[1]{{\texttt{\textbackslash#1}}}
```

Sometimes we need to change the definition of existing commands. This is accomplished using the command `\renewcommand`. Its syntax is the same as that for `\newcommand`, except in that the command name is no longer one we make up, but instead is the name of an existing command. We already saw this in action in Section 3.5:

```
\renewcommand{\sfdefault}{phv}
```

This redefined the existing sans-serif default font to one associated with the name "phv". Another place where we saw this in action was in the macro that specifies the space between lines:

```
\renewcommand{\baselinestretch}{2.0}
```

would double the spacing from one line to the next.

In order to change formatting on long bodies of text, we typically need to impose the formatting changes at the beginning of the text, and then undo those changes at the end. In other words, we need to make an environment. Obviously, LaTeX provides a command to let us do that. The following command defines an environment that contains indented text.

```
\newenvironment{inden}
{\hfill\begin{minipage}{.9\textwidth}}
{\end{minipage}}
```

This illustrates most of the critical features of the `\newenvironment` command. The command takes three arguments: the name of the environment we want to define; the commands that will be used at the start of the environment, and those used at the end. Thus, to invoke the environment defined above, we would type `\begin{inden}`. When that happens then the new environment starts a minipage, that is only 90% of the page width, with horizontal space to indent it. This is how most new environments we define work: We build them off existing environments, and add few commands to modify those somewhat.

Once again, we can change an existing environment using the `\renewenvironment` command. It works in the obvious way.

Chapter 4

Computers

We have discussed networking and the World Wide Web. We have engaged in computer typesetting, and the display of documents in many ways. Nonetheless, we have not yet engaged with the computers themselves – we have only learned how to run individual programs on those machines. The fact is that this is how we always use computers – we *only* execute programs on them. Most people never interact with machines on a low level. Instead we just double-click some icon that represents a program for the computer, and then interact with that.

The nice thing about running other peoples' programs is that we benefit from their expertise and hard work. On the other hand, when we run other peoples' programs, then we can only do what they let us do. When we use a web browser, we can read email, we can read informative pages, but we can probably not solve our math homework problem (well, using Wolfram Alpha [2], we *might* be able to solve some of our homework problems). On the other hand, computers can do almost anything short of rustling up a ham sammich for us, if we know how to instruct them to do that. We will never interact with the computers directly, i.e. in machine language, but from now on we aim to work with the computers on a more command-driven level. In that way we can have greater control over them.

Even as we do this, we will still be working in programs other people have written for us. However, these programs will be designed to give us greater access to all the capabilities of the computers we use. Instead of simply making use of the machine's ability e.g. to fetch and display text and images, we will cause the computer to crunch numbers, create those images, and solve our homework problems for us. The cost to us is learning a bit about some arcane languages.

4.1 Using a command line

One of the most basic programs for a computer is called a *shell*. A shell is a basic command line interpreter. Its job is to provide a somewhat natural syntax for giving commands to a computer – more natural than the computer's machine instructions. Moreover, with a common command set, it is possible for a person familiar with the syntax to interact with computers that have many different architectures and instruction sets. The shell does the translation into machine language for us.

Command line shells are considered to be very primitive interfaces to computers. Most people have a bad reaction when told they must use command lines. This might result from the obscurity of some of the commands, or it might simply be a knowledge that command lines are one of the oldest and most primitive ways to interact with computers. On the other hand, command line shells use very little overhead, and have comprehensive instruction sets. We can do anything using command line shells, all while requiring very little network bandwidth. The fact is that the most sophisticated users of computers use command line interfaces almost exclusively, for those reasons. Scientists and mathematicians, in particular, often find themselves working on powerful computers far from where they sit. In that situation, a command line shell is the fastest way to tell those powerful computers what to do. For that reason, we should learn at least a few useful commands, and some of the characteristics of the environment in which they run.

When we speak of powerful computers, we ordinarily mean high performance clusters. These are machines typically with hundreds or thousands of processors. The jobs we run on those machines might take minute, hours, or even days, and will almost certainly require a lot of resources – far beyond what the ordinary desktop computer has. Such machines use some modern variation of Unix. Indeed, as of this writing, almost all such machines use some version of Linux, and these machines almost always use the Bash shell.

One of the first command line shells was called the Bourne shell, named after Stephen Bourne who invented it at Bell Labs. Since it was early, it became a standard shell for Unix systems. Unfortunately, it was tightly tied to the proprietary UNIX operating system (as opposed to Unix considered as a class of similar operating systems). For that reason, when the Gnu project needed a free shell, the Bourne shell was not used. Instead, its command structure was duplicated and augmented. The resulting shell was called the "Bourne again shell", or Bash for short. This has become the most commonly-used shell on all computers that use Gnu, including all distributions of Linux, Apple Macintosh computers, and others. Our goal here is not to learn all the details of the Bash shell. We will simply learn a few basic Bash commands and some essential information that we can use when we need to.

The basic format of Bash shell commands goes like this:

```
command -options arguments
```

Thus, we give a command name first. After that, we type various types of input data for the command. All input data are separated by spaces. Command options are usually preceded by a hyphen, though it is traditional to call it "minus". Many commands operate on file names and directories. These are typed in full as arguments.

4.1.1 Directory commands

One of the most basic Bash commands one uses prints the working directory name. The command is `pwd`. This is typical of Unix commands: relatively few commands are longer than four or five characters. The Bash shell has a command-completion feature: one can hit the <Tab> key any time, and Bash will try to complete the command name or the file name you are typing. Thus, nowadays long command names are easier to work with – the typical Bash user always has a finger near the <Tab> key. However, earlier Unix shells did not have this feature, so there was a strong effort to keep commands short.

Once we know which directory we are in, we probably want to list all the files in it. To do this, we just type `ls`. Note that "list" would be too long – the command has only two characters. This command is tremendously useful, and consequently has many options. In the simplest setting we might see something like this:

```
$ pwd
/home/username
$ ls
test.aux  test.log  test.pdf  test.tex
```

Note that we do not type the $, that is a *prompt* provided by Bash. We only typed the commands after the $. Now we might want to know more about those files, for example, their size,and the dates on which they were last modified. For this, the `ls` command provides the `-l` option.

```
$ ls -l
drwxr-xr-x. 2 username grp  4096 Oct 14 11:10 subdir
-rw-r--r--. 1 username grp     8 Oct 14 10:37 test.aux
-rw-r--r--. 1 username grp  3486 Oct 14 10:37 test.log
-rw-r-----. 1 username grp 48410 Oct 14 10:37 test.pdf
-rw-r-----. 1 username grp  2159 Oct 14 10:37 test.tex
```

The `-l` stands for "long format". The output requires some explanation. The first collection of characters describes the permissions on each file. The first character of this string tells whether the file is a directory "d", an ordinary file "-", or some other kind of file. After that comes three sets of three characters describing the permissions of the owner of the file, the group that controls the file, and of other users. In each subgroup of three characters, "r" stands for read permission: the user can examine the file or copy it. The "w" stands for write permission: the user can modify or delete the file. The "x" stands for execute permission: the user can actually run the file. Naturally, for this last to work, the file must be executable - it must be a program, and not just text or data.

After the permission string comes the account name that owns the file, followed by the group associated with the group permissions. Next comes the file size in blocks – more on this later. This is followed by the date of the last modification to the file, and finally, the file name.

In the listing above, we see that `subdir` is a directory. That subdirectory allows execute permission for the owner, the group, and for all other users. This is normal – execute permission on a directory is required just to list its contents. The other files all allow read and write permission to their owner, and only read permission for everyone else. The files `test.tex` and `test.pdf` allow read permission for its group, but other users do not have permission to do anything with them. Most users can look at the files' names and drool – they cannot copy them or even view them.

The file size was given in blocks. This is an alarmingly useless way to list the file size – it tells us which files are larger than others, but really does not help us find out whether a file will fit on our thumb drive. The "-h" option gives a human-readable format; i.e. kilobytes, megabytes, and so on. The "-a" option shows all files, including hidden ones.

```
$ ls -lah
drwxr-xr-x.  3 kcooper man 4.0K Oct 14 11:10 .
drwxr-xr-x. 27 kcooper man  24K Oct 14 10:36 ..
drwxr-xr-x.  2 kcooper man 4.0K Oct 14 11:10 subdir
-rw-r--r--.  1 kcooper man    8 Oct 14 10:37 test.aux
-rw-r--r--.  1 kcooper man 3.5K Oct 14 10:37 test.log
-rw-r--r--.  1 kcooper man  48K Oct 14 10:37 test.pdf
-rw-r--r--.  1 kcooper man 2.2K Oct 14 10:37 test.tex
```

In this case we see that `test.tex` only contains 2.2 kilobytes of data, while `test.pdf` contains 48 kilobytes.

We see also two hidden directories in this listing. The first is called ".". The single period always refers to the current directory. In Unix, every directory, every device, everything is labeled as a file, so "." is the file that points to the current directory. In a similar way, ".." is a file that points to the parent of the current directory.

Unix always considers that we are "in" a directory. When our shell starts, we are "in" our home directory. When we use the `ls` command without a file name then that command shows us the listing of the directory we are "in" – our current working directory. If we want to see the contents of some other directory, we can give the name of that as an argument to `ls`.

```
$ ls -lh subdir
-rw-r--r--. 1 kcooper man    0 Oct 14 13:50 oldtest.tex
drwxr-xr-x. 2 kcooper man 4.0K Oct 14 13:53 subsubdir
```

When we chase through directories this way, the names of the directories are separated by forward-slashes (/). Thus, if we now want to display the contents of the `subsubdir` directory, we can type

```
$ ls -la subdir/subsubdir
drwxr-xr-x. 2 kcooper man 4096 Oct 14 13:54 .
drwxr-xr-x. 3 kcooper man 4096 Oct 14 13:53 ..
-rw-r--r--. 1 kcooper man    0 Oct 14 13:54 bottomfeeder.tex
```

If we actually need to change our current working directory, we can use the `cd` command – change directory.

```
$ cd subdir
$ pwd
/home/username/subdir
```

Notice that when `pwd` gives the directory, it gives the *absolute path* to it. It starts at the unique top level directory, called "/", and then tracks down through subdirectories to get to the current directory. Thus `pwd` above says that we are in the `subdir` subdirectory of the `username` directory, which is a subdirectory of the `home` subdirectory of the root (/) directory. We can also change directories using this notation.

```
$ cd /home/username/subdir/subsubdir/
$ pwd
/home/username/subdir/subsubdir
```

We would ordinarily only use this format when we are changing to a directory far from our current one.

Using the `cd` command with no arguments always takes you to your home directory.

```
$ cd
$ pwd
/home/username
```

We can make a new directory in any location where we have write permission by using the `mkdir` command.

```
$ mkdir mynewdir
$ cd mynewdir
$ pwd
/home/username/mynewdir
```

We can delete an empty directory using the `rmdir` command, but since directories are almost never empty, we won't really trouble ourselves with that – there are better ways to go about it.

4.1.2 File commands

Once we arrive in a working directory where our project files reside, we'll need to copy, move, and alter them. The simplest command for moving files around is `cp` – copy. We can copy e.g. `test.tex` to a new file called `newtest.tex` as follows.

```
$ ls
mynewdir  subdir  test.aux  test.log  test.pdf  test.tex
$ cp test.tex newtest.tex
$ ls
mynewdir     subdir    test.log  test.tex
newtest.tex  test.aux  test.pdf
```

We see that the directory contains one new file called `newtest.tex`. Suppose now that we realize seconds after the `cp` command that we actually wanted to make the new file in the `mynewdir` subdirectory. We can move the new file there using the `mv` command – move.

```
$ mv newtest.tex mynewdir
$ ls mynewdir
newtest.tex
$ ls
mynewdir  subdir  test.aux  test.log  test.pdf  test.tex
```

Note that we did not need to type the name of the file in the new directory – when we left it off, then Bash assumed it should keep the same name. We would only need to type a name of the file if we wanted to change that name. That works for directories as well. The `mv` is what we use to change the names of files and directories.

```
$ ls
mynewdir  subdir  test.aux  test.log  test.pdf  test.tex
$ mv mynewdir newfiles
$ ls
newfiles  subdir  test.aux  test.log  test.pdf  test.tex
```

We could copy the `newtest.tex` file back to the current directory fairly simply.

```
$ cp newfiles/newtest.tex .
$ ls
newfiles     subdir    test.log  test.tex
newtest.tex  test.aux  test.pdf
```

We can remove files using the `rm` command.

```
$ rm newtest.tex
$ ls
newfiles  subdir  test.aux  test.log  test.pdf  test.tex
```

We can type the names of all the files we want to remove on the same command line. It is happy to remove one, or ten, as we like. Since it is so happy to remove files for us, sometimes we might want to be careful. We might ask the `rm` command to inquire as to whether we really want to remove the file before it goes through with that operation.

```
$ cp test.tex newtest.tex
$ rm -i newtest.tex
rm: remove regular file ‘newtest.tex’? y
$ ls
newfiles  subdir  test.aux  test.log  test.pdf  test.tex
```

We typed the “y” in response to the question in the middle.

We can remove entire directories using the `rm -r` command. The “-r” option stands for “recursive”, i.e. it follows the directory structure down recursively. If there are not many files in the directory, we should inquire about every file.

```
$ rm -ri newfiles
rm: descend into directory ‘newfiles’? y
rm: remove regular file ‘newfiles/newtest.tex’? y
rm: remove directory ‘newfiles’? y
```

If we are certain about the contents of the directory, we can use the “-f” option instead of “-i”. the “-f” stands for “force”, so that in this case `rm` does not ask about anything – it just destroys it. Needless to say, we use this with caution.

```
rm -rf newfiles
```

4.2 SSH

4.3 SFTP

4.4 Computer numbers

Chapter 5

Matlab

In this chapter we discuss one of the most popular computer programs for performing mathematical operations. Several such programs have established themselves in recent years. These all came from different roots, but have strongly overlapping capabilities now. Let's take a brief overview of the chief suspects.

The first program for doing symbolic mathematics using a computer was called Macsyma (Machine Symbolic Manipulator), and was developed at the Massachusetts Institute for Technology in the early part of the 1980s. It did what its name implied: limited manipulation of mathematical expressions. Initially it had no graphics capability. At the same time, a number of programs were being developed for non-symbolic aspects of mathematics and statistics. For example, Minitab was an early leader in statistical computing that could also plot data in a variety of formats. Later in that decade, other programs came onto the scene. By the beginning of the 1990s, several programs had assumed positions of prominence in the world of computer mathematics: Matlab, Maple, Mathcad, and Mathematica. In the world of statistics, several other programs had become popular: in particular, S-Plus and SAS had drawn some of Minitab's market away. More recently, Octave has become a very good free alternative to Matlab, and R is a fine free alternative to S-Plus.

Matlab was designed from the start as a means for manipulating matrices and vectors easily, with standard linear algebra tools, such as LU decomposition, built in. It only brought symbolic capabilities in later, and that by purchasing tools from Maple. Minitab is similar to Matlab, except in that the emphasis is on statistics instead of linear algebra. By contrast, Maple and Mathematica both followed in Macsyma's footsteps, and competed head-to-head. Mathematica was developed initially by the physicist Stephen Wolfram. Its syntax reflected, to some extent, the disciplinary bias of its author. Maple was developed by a consortium of mathematicians and computer scientists at the University of Waterloo in Canada, and its syntax thus followed more closely the traditional notation of mathematics. Mathcad grew differently, emerging as a free-form mathematical spreadsheet program that employed a more natural notation and a more extensive function library than comparable Microsoft and Borland products. Later, it also incorporated some symbolic capability derived from Maple.

In the beginning of the twenty-first century all of these programs can perform many of the same tasks, but some excel in certain applications. They can all solve simple systems of algebraic equations; they can all compute derivatives and antiderivatives of functions; they can all multiply matrices, and solve matrix equations; and they can all plot functions in a variety of ways. They are all capable of writing their output in a variety of formats, including HTML and TeX. On the other hand, most users would accept the proposition that Matlab is the best at handling numerical computations, while Maple and Mathematica are superior for symbolic calculations. Mathcad is marvelous for small-scale parametric modeling - the kind of ”what if” calculations done for simple models or in the early stages of mathematical investigations.

5.1 Matlab Basics

The program Matlab dates from the mid-1980s. It came out of work by Cleve Moler, who remains the head of the company that produces it today [1]. It was designed with relatively humble objectives: to provide a simple interface to high-quality numerical procedures for solving mathematical problems. Indeed, Matlab stands for Matrix Laboratory. It has effectively become the standard package for mathematicians and engineers who need to do quick-and-dirty estimates, simulations, designs, and tests of algorithms.

There are two basic versions of Matlab: the full version, and the student version. The student version is actually fairly complete, but lacks some of the plotting capabilities that the full version has. On the other hand, the cost of the student version is dramatically lower than that of the full version. With the full version, one may purchase additional “toolboxes” – collections of functions designed for specialized tasks such as analysis of partial differential equations, spline approximation, and the like.

When you start Matlab, regardless of the details of the interface, you should notice immediately that it is basically just a command line. You type commands, hit <Enter>, and the commands are executed. The basic syntax is very natural to anyone who has used a computer before. For example, to do a simple arithmetic calculation, just type it in and press <Enter>.

```
>> 3+4

ans =

     7
```

This illustrates several basic notions behind Matlab. First, Matlab can do all basic arithmetic operations with a very natural syntax. Second, Matlab requires that every computation be assigned to a variable. If you do not specify the name of the variable, Matlab uses a default name: `ans`. Matlab reuses the name `ans` as needed, so if you want to keep the result of a computation for reference later in a session, you should assign it to a different variable. For example,

```
>> seven=3+4

seven =

     7
```

assigns the value 7 to a variable called `seven`. We may view the contents of the variable seven at any time simply by typing its name.

```
>> seven

seven =

     7
```

One may use variables interchangeably with the numbers or objects that they represent. For example, we could define a variable called `ten` by

```
>> ten=seven+3

ten =

    10
```

Matlab uses "+" to denote addition, "-" to denote subtraction, "*" for multiplication, "/" for division, and " ˆ " to denote exponentiation. Thus,

```
>> myvariable=2*(ten-2)+(ten/5)^2

myvariable =

    20
```

Of course, arithmetic is not the only thing that Matlab can do. As we mentioned earlier, it was designed with linear algebra in mind. Thus, it excels in dealing with vectors and matrices. Vectors and matrices are defined using [brackets] to enclose them.

```
>>row_vector=[1 2 3]

row_vector =

     1     2     3
```

One may equally well separate the entries in the vector using commas:

```
>>row_vector=[1,2,3]

row_vector =

     1     2     3
```

To define a column vector, one may type the entries on separate lines.

```
>>col_vector=[1
2
3]

col_vector =

     1
     2
     3
```

If you prefer not to waste so much space in entering a column vector, you may separate the rows of the vector using semicolons. Semicolons always denote the end of a line of entries.

```
>>col_vector=[1;2;3]

col_vector =

     1
     2
     3
```

You might also get a column vector by taking the transpose of a row vector. Matlab uses an apostrophe ' to denote the transpose of any matrix or vector.

```
>>col_vector=[1 2 3]'

col_vector =

     1
     2
     3
```

One types matrices into Matlab analogously to the way one enters vectors. Indeed, Matlab does not distinguish between matrices and vectors – it treats vectors as $n \times 1$ or $1 \times n$ matrices. Thus we might enter a 2×3 matrix in the following way.

```
>> A=[1 0 0
0 1 0]
```

```
A =

     1     0     0
     0     1     0
```

or alternatively as

```
>> A=[1 0 0; 0 1 0]

A =

     1     0     0
     0     1     0
```

Matlab handles matrix arithmetic just as easily as scalar arithmetic. The symbols for the operations are the same, but their meaning is changed to that appropriate for matrix operations. For example,

```
>> A*col_vector

ans =

     1
     2
```

Matlab keeps track of the dimension of the matrices it is handling, and refuses to perform illegal matrix operations. Remember that A is a 2×3 matrix.

```
>> A*[1 2 3]
??? Error using ==> *
Inner matrix dimensions must agree.
```

By now you have noticed that Matlab always prints out the result of a computation. If you type only the name of a single variable, then Matlab prints the value of that variable. Sometimes you will not want to see the result of a computation. You may suppress output on any line by ending it with a semicolon. For example

```
>> [1 2];
>> [2 2]

ans =

     2     2
```

While numerical analysts will tell you that you should never compute an inverse (it is almost always more work than simply solving a matrix equation), Matlab has no problem in doing so. To compute the inverse of a matrix A we must invoke a built-in Matlab function called inv(). For example,

```
>> A=[1 0
-1 1]

A =

     1     0
    -1     1

>> x=inv(A)*[1 1]'

x =

     1
     2
```

Of course, if we had used the 2×3 A that we had earlier, the `inv( )` function would have given an error.

The reason we should probably never use the `inv` function is that doing an L-U decomposition and back substitution is essentially always less work if you need to find the solution to a linear system. If you are not familiar with that terminology, it effectively means simple row reduction, also called Gauss elimination. Matlab uses the backslash "\" to solve matrix equations. We can arrive at the solution above using the commands below.

```
x = A\[1 1]'

x =

     1
     2
```

In other words, the syntax `A\[1 1]'` tells Matlab to compute the solution x to the equation

$$Ax = \begin{pmatrix} 1 \\ 1 \end{pmatrix}.$$

Example 5.1 Consider the salmon data from the example chapter. We want to use Matlab and a simple least-squares process to fit a line to the control group of those data. First, we type in the data vector.

```
>> y=[4.2 8.5 8.6 9.4 9.8 10.3 11.2 15.5]';
```

Next, we need to form basis vectors for the space from which we intend to approximate the data. Since we want an approximation that is a polynomial of degree one, we shall use functions f1(x) = 1 and f2(x) = x to generate vectors that we use for the approximation. We must evaluate these functions at the x-values for the data to get the vectors. It is easy to see that one vector will be composed simply of ones, while the other will have entries that are the days on which the data were taken. We use the commands below.

```
>> v1(1:8)=1;
>> v2=[-36 0 4 16 26 49 80 208]';
```

We make a matrix A to use in solving for the coefficients of the approximation out of the column vectors we have just defined.

```
>> A=[v1 v2];
```

Next, we solve for the coefficients.

```
>> coeffs=(A'*A)\(A'*y)

coeffs =

    7.9732
    0.0395
```

We can return the values of the line we have found at the `x` data points simply by multiplying the coefficients by `A`.

```
>> line=A*coeffs

line =

    6.5505
    7.9732
    8.1313
    8.6056
    9.0008
    9.9098
   11.1350
   16.1938
```

Note that Matlab is case sensitive. Thus, there are no functions called `Inv( )` or `INV( )`, and a variable called “a” would be different from “A”.

```
>> A

A =

     1     0
    -1     1

>> a
??? Undefined function or variable 'a'.
```

There is no function to compute the dot product of vectors - it is not necessary. If x and y are column-vectors of the same dimension, then their dot product is found as `x'*y` or `y'*x`. When Matlab displays results, the it uses four significant digits by default. It actually performs computations in double precision arithmetic, but displays the numbers using lower precision. If you should want to see more precision, you may use the long format. For example,

```
>> z=2*pi

z =

    6.2832

>> format long
>> z

z =

   6.28318530717959

>> format short
>> z

z =

    6.2832
```

Observe also that Matlab has predefined values for a variable called `pi`, which of course is an approximation to the famous trigonometric constant π.

Often we want to have Matlab compute some large matrix that we will work with, but we do not need to look at it. In that case, Matlab's practice of echoing every result becomes troublesome – it is unnecessary and slow. We can suppress that output by typing a semicolon at the end of a line.

```
>> A=[1 2; 3 4; 5 6]

A =

     1     2
     3     4
     5     6

>> A=[1 2; 3 4; 5 6];
>>
```

Note then that there are two uses of the semicolon. Whenever we insert a semicolon, it ends a line. However, when we use it at the end of a Matlab command, it also suppresses the output of that command.

The “Matlab Commands” reference summarizes some of the most useful Matlab commands.

5.2 Matrices

We have typed vectors and matrices, but as these become larger typing each entry becomes tedious, burdensome, and prone to errors. Instead we should be able to create and manipulate matrices with greater ease. We must be able to work on small parts of matrices, change individual elements, or build them one element at a time, if necessary. Matlab can do all of that.

Consider the 4×4 matrix A given by

$$A = \begin{pmatrix} \cos(\pi/6) & -\sin(\pi/6) & 0 & 0 \\ \sin(\pi/6) & \cos(\pi/6) & 0 & 0 \\ 0 & 0 & 1 & 0 \\ 0 & 0 & 0 & 1 \end{pmatrix}.$$

We can create an entire matrix of zeros by using the `zeros()` function. This makes a matrix of zeros in the dimension given in the argument. The following command makes a 4×4 matrix of zeros. Remember that the semicolon at the end of the line suppresses the output. We really don’t need to see that Matlab made all those zeros.

```
>> A = zeros(4);
```

That done, we can address any element of the matrix A using indices in parentheses. The idea is that the Matlab notation for $A_{i,j}$, the element of A in the i^{th} row and j^{th} column, is `A(i,j)`. To emphasize, every entry $A_{i,j}$ corresponds in Matlab to `A(i,j)`. Thus, we can set the last two diagonal elements of the Matlab matrix `A` to one using the commands

```
>> A(3,3) = 1;
>> A(4,4) = 1

A =

     0     0     0     0
     0     0     0     0
     0     0     1     0
     0     0     0     1
```

We suppressed the output from setting $A_{3,3}$, but allowed A to print out after setting $A_{4,4}$, just to keep track of what we have done.

Using range notation (i.e. using colons), we can assign entire submatrices of A. To put the 2×2 rotation matrix in the upper left corner of A, we can use the lines

Reference 5.1

Matlab Commands

The following are some of the simpler and more commonly used commands in Matlab. We have not provided details of their use. Indeed, most commands may be used in a variety of ways. See the help pages for more information.

`abs(x)` - returns the absolute value of the entries of a matrix x.

`cond(x)` - returns the 2-norm condition number of the matrix x.

`det(x)` - returns the determinant of the matrix x.

`eig(x)` - returns a vector containing the eigenvalues of the matrix x. `[v,d]=eig(x)` returns a matrix `d` whose diagonal entries are eigenvalues, and a matrix `v` whose columns are eigenvectors of x.

`eye(n)` - returns an $n \times n$ identity matrix

`format long` - causes numbers to be displayed in full-precision.

`format short` - causes numbers to be displayed with only five digits of precision.

`inv(x)` - returns the inverse of the matrix x, if it exists.

`mesh(x)` - plots a mesh surface of the values in the matrix x.

`norm(x)` - returns the norm of the matrix x.

`ones(n)` - returns an $n \times n$ matrix of ones. Likewise, `ones(m,n)` returns an $m \times n$ matrix of ones.

`plot(x,y)` - creates a graph of y against x.

`plot3(x,y,z)` - creates a graph that is a projection of the segments formed by joining successive points (x_i, y_i, z_i), where x_i is the ith element of the vector x, etc.

`rand(n,m)` - creates an $n \times m$ matrix of uniformly distributed pseudo-random numbers in $[0, 1]$. This matrix is typically seeded. You can randomize it by first calling `rng()`.

`rank(x)` - returns the rank of the matrix x.

`size(x)` - returns the dimension of the matrix x.

`zeros(n)` - returns an $n \times n$ matrix of zeros. Likewise `zeros(m,n)` returns an $m \times n$ matrix of zeros.

```
>> p6=pi/6;
>> A(1:2,1:2)=[cos(p6) -sin(p6); sin(p6) cos(p6)]

A =

    0.8660   -0.5000         0         0
    0.5000    0.8660         0         0
         0         0    1.0000         0
         0         0         0    1.0000
```

Let's do this again, on another matrix. Consider

$$B = \begin{pmatrix} 1 & -2 & 1 & 0 & 0 & 0 \\ 0 & 1 & -2 & 1 & 0 & 0 \\ 0 & 0 & 1 & -2 & 1 & 0 \\ 0 & 0 & 0 & 1 & -2 & 1 \end{pmatrix}$$

We again initialize a matrix of the appropriate dimension to zero. This time, since the matrix is not square, we must use the `zeros()` function with two arguments. There are several functions in Matlab that work like this: they have one behavior with one argument, a different behavior with two arguments, and perhaps other behaviors as we use more and more arguments. In this case, zeros makes a square matrix if we give it a single integer argument, and it makes a matrix of dimension m, n if we give it two arguments `m,n`. Once we make the matrix of zeros, we make a vector to hold that pattern $1, -2, 1$, and then set that into the appropriate submatrix on each row.

```
>> B=zeros(4,6);
>> pattern=[1 -2 1];
>> B(1,1:3)=pattern;
>> B(2,2:4)=pattern;
>> B(3,3:5)=pattern;
>> B(4,4:6)=pattern

B =

     1    -2     1     0     0     0
     0     1    -2     1     0     0
     0     0     1    -2     1     0
     0     0     0     1    -2     1
```

We could now pick out the middle part of that matrix to get a 4×4 nonsingular matrix.

```
>> C=B(:,2:5)

C =
    -2     1     0     0
     1    -2     1     0
     0     1    -2     1
     0     0     1    -2
```

We see that the notation leaving a colon as a placeholder in the position of one index means: "just get all of the possible values here." There are a few interesting things that can be done with this notation, but probably the most useful makes use of the fact that in a one-dimensional vector v, `v(:)` is always a column vector. In other words, the default form for a vector that constitutes all the elements of an array is a column. We can use this sometimes when we do not know the format of a vector to force it into a form that we know.

```
>> v=[1 0 -1];
>> v(:)

ans =

     1
     0
    -1

>> v(:)'

ans =

     1     0    -1
```

There are many ways to initialize matrices – `zeros()` is not the only one. In particular, there is a function `eye()` generates an identity matrix of dimension as specified in its single argument. Using this we could have created the matrix A earlier in this section using the following commands.

```
>> A = eye(4);
>> A(1:2,1:2)=[cos(pi/6) -sin(pi/6); sin(pi/6) cos(pi/6)]

A =

    0.8660   -0.5000         0         0
    0.5000    0.8660         0         0
         0         0    1.0000         0
         0         0         0    1.0000
```

As it happens, in Matlab we do not even have to initialize matrices. Instead, we can just start assigning values. Matlab increases the size of the matrix to accommodate the elements we assign. Note that this process of enlarging the matrix is very slow – this is a bad way to do things. Nonetheless, it illustrates the flexibility of Matlab's data structures.

```
>> U(1,1)=1; U(1,2)=2; U(2,1)=3; U(2,2)=4;
>> U

U =

     1     2
     3     4
```

Matlab views vectors as $n \times 1$ or $1 \times n$ matrices. Thus, you could refer to elements of the vector v that we defined above using two subscripts.

```
>> v(2)

ans =

    0.0000

>> v(1,2)

ans =

    0.0000
```

By the same token, when Matlab stores matrices, it treats them as long vectors in row-major order; i.e. it acts as if the matrix is actually a long vector in memory, starting with the first row, followed by the second row, and so on. In a pinch we can refer to elements of matrices in this sequential way. Consider the matrix A we have been working with.

```
>> A(5)

ans =

   -0.5000

>> A(6)

ans =

    0.8660
```

It is important to observe that the notation for subscripted elements is the same as that for arguments to functions. Matlab keeps track of which objects are functions and which are matrices. It is important for you to do so also. Moreover, you must give both your variables and functions names that do not conflict with predefined Matlab functions – do not name your matrices `sin` or `exp`.

5.3 Elementwise Operations

We have seen that Matlab interprets arithmetic operations in the context of matrix arithmetic. On the other hand, very often we want to create vectors of values and then manipulate the entries of those vectors individually. Matlab provides many tools to make this easy.

Suppose that we want to create a vector of integers from 1 to 10. To do so, we only need to type variable=first:last, where first is the value of the first entry in the vector, and last is an upper bound for the last entry. The default increment for each entry is one.

```
>> x=1:10

x =
     1     2     3     4     5     6     7     8     9    10
```

The first entry does not have to be an integer, and the difference between the first and the last is not required to be an integer. If the difference of the first and last numbers is not an integer, then the largest entry in the vector is the first plus the greatest integer smaller than the difference of the last and first. It is easier to see in an example.

```
>> x=.83:9.8

x =

  Columns 1 through 7

    0.8300    1.8300    2.8300    3.8300    4.8300    5.8300    6.8300

  Columns 8 through 9

    7.8300    8.8300
```

Often we want to create a sequence of points with some uniform spacing other than one. That calls for another term in the definition for the vector.

```
>> x=1:.5:10

x =

  Columns 1 through 7

    1.0000    1.5000    2.0000    2.5000    3.0000    3.5000    4.0000

  Columns 8 through 14

    4.5000    5.0000    5.5000    6.0000    6.5000    7.0000    7.5000
  Columns 15 through 19

    8.0000    8.5000    9.0000    9.5000   10.0000
```

Note that the result was a row vector of dimension 19. Since Matlab cannot put all 19 entries on a single line, it uses several rows, and labels the columns so that we can understand what we are looking at in the vector.

Once we have a vector of values, we may manipulate it entry by entry in a variety of ways. First, we note that Matlab has many functions that apply naturally to scalars. When vectors are given as arguments to those functions, they apply elementwise. For example,

```
>> x=0:.5:pi

x =

         0    0.5000    1.0000    1.5000    2.0000    2.5000    3.0000

>> sin(x)

ans =

         0    0.4794    0.8415    0.9975    0.9093    0.5985    0.1411
```

Naturally, Matlab provides all of the other trigonometric, logarithmic, and exponential functions you would expect, i.e. `tan(x)`, `cos(x)`, `exp(x)`, and `log(x)` are all valid. On the other hand, algebraic operations are a little less transparent. For example, Matlab does not understand how to compute the square of a vector, hence if we want to compute the square of each element of the vector, we need a different notation. Matlab provides this by preceding any operation you need by a period (“ . ”). Thus

```
>> x.^ 2

ans =

         0    0.2500    1.0000    2.2500    4.0000    6.2500    9.0000

>> x.*x

ans =

         0    0.2500    1.0000    2.2500    4.0000    6.2500    9.0000
```

This is very useful, and as we shall see later, quite important. These operations are fast. We could address elements in the vector one at a time as we did in the previous section, but that is much slower.

5.4 Matlab Input and Output

Input and Output, known in shorthand as I/O (pronounced “eye-oh”), is an essential part of any computer program. It is a particularly powerful and useful component of Matlab, inasmuch as the package is frequently used to make complicated plots of data computed in programs or measured in the field.

5.5 Matlab Plots

Matlab provides very powerful tools for plotting functions, data, surfaces, and almost anything else you need. It can save its graphics in a variety of formats, including postscript, bitmaps, PNG, JPEG, and GIF images. Thus, it is ideal for creating graphics for publications or for the Web. There are some caveats in order here. The student version of Matlab does not provide the ability to save plots in graphic formats - indeed, it scarcely makes provision for saving graphics at all.

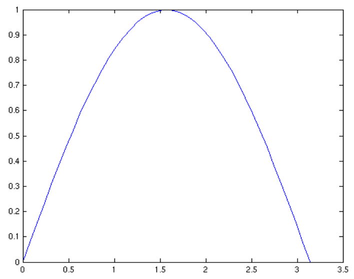


Figure 5.1: A simple sine curve in Matlab

The basic tool we use in creating graphics is the plot command. For example, suppose that we have defined `x` to be a vector of equally spaced points on the interval [0,3.14] and `y=sin(x)`. Then we may plot those using the command `plot(x,y)`. This simple command yields the picture in Figure 5.1.

The pair of vectors containing the abscissæ and ordinates for the plot is called a *data series.* The `plot` command can take essentially any number of data series. In this way, the `plot` function has the interesting property of accepting a variable number of arguments – this is not true for most Matlab commands.

If one gives only two arguments for a data series in a Matlab plot, it appears as a line connecting the data points, in a default color. When more than one data series is specified without formatting, each successive data series appears in a different color from the default choices. One way in which you might need to customize the appearance of the plot is through the line or point style used for a data series. For example, very often one needs to plot data points as well as a curve that interpolates or approximates them. The Matlab `plot` command accepts an optional third argument for each data series that allows you to specify the line or point style, and the color. Thus, we see that specifying a data series in `plot` can be done using either two arguments, giving abscissæ and ordinates of the points to be plotted, or three: giving the abscissæ, ordinates, and some simple graphical instructions. Matlab figures all of this out.

Reference 5.2

Matlab Plot Format Strings

There are a number of easy ways to customize the appearance of individual data series in Matlab through the associated format strings – the optional third argument for the data series.

Colors

The set of colors available by default follows.

b	blue
g	green
r	red
c	cyan
m	magenta
y	yellow
k	black
w	white

Point Styles

Point styles in the format string can vary from none to elaborate hexagrams. The default is not to show the points, but instead simply to connect the points by line segments. One can mix the point styles with the line styles; indeed if one specifies a point style, the default is not to show lines, so if you want both points and lines, you must specify both.

.	show points as tiny filled circles
o	show points as small unfilled circles
x	show points as small crosses
+	show points as plus symbols
s	show points as small unfilled squares
d	show points as small unfilled diamonds
v	show points as downward-pointed triangles
p	show points as five-pointed stars
h	show points as six-pointed stars
*****	show points as eight-pointed stars

Line Styles

By default data series are displayed using solid line segments joining points that are not displayed.

-	show lines as solid segments
–	show lines as dashed segments
:	show lines as dotted segments
-.	show lines as dashes alternating with dots

To use the format strings, one just combines the symbols specifying color, point, and line styles. For example, we could plot the data points $(0, 4.3)$, $(21, 5.6)$, $(26, 5.2)$, and $(36, 6.1)$ as follows.

```
plot([0 21 26 36],[4.3 5.6 5.2 6.1],'mh-.')
```

Obviously there are other ways in which you might want to customize the appearance of this plot. For example, you should label the axes, and perhaps give it a title. That may be done in either of two ways. You might choose to use the following commands to make the labels.

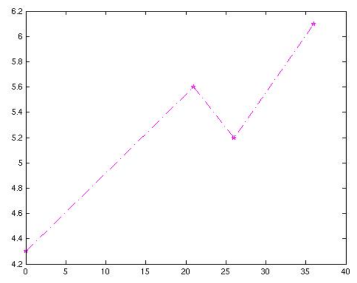


Figure 5.2: A simple formatted plot

```
>> xlabel('x values')
>> ylabel('sin(x)')
>> title('Demonstration Plot')
```

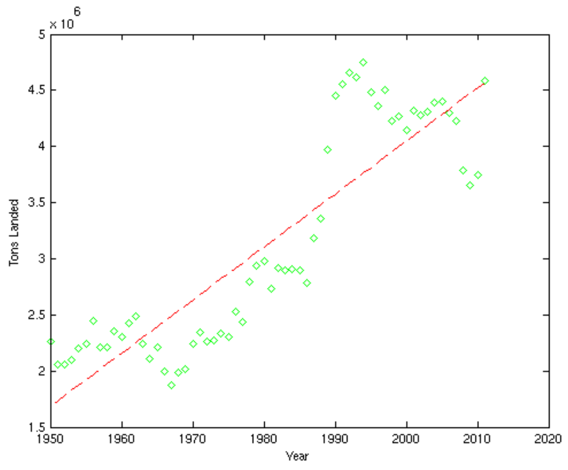
Recall that Matlab uses single-quotes to enclosed text strings. The quotes do not show on the graphic.

Alternatively, you could choose to apply the labels directly to the figure through the menu provided on the window containing it. This option again depends on which version of Matlab you use. If you have the student version, then the graphic appears in a window that does not allow you to do much besides print it. On the other hand, if you have the full version of Matlab, then there is a menu that allows you to customize any aspect of the plot.

Example 5.2 We created a curve to fit the data for the commercial fish landings from Section one of this chapter. Now we want to plot the results. We plot the data points as green diamonds, while the line that approximates them appears as in red, and is dashed. We also labeled the axes.

```
>> plot(x,y,'gd',x,line,'r--');
>> xlabel('Year');
>> ylabel('Tons Landed');
```

The plot that results appears below.



Another way to create a more complicated figure is to superimpose one type of plot on another. This is done simply by making the first plot, and then using the hold command:

```
>> plot(x,y);
>> hold on
>> plot(x,z,'rd')
```

One usually uses this approach when mixing types of plots. For example one could superimpose a patch plot of polygons on a normal plot of a function.

We will discuss more advanced topics in the creation of graphics using Matlab later in this chapter.

5.6 Matlab Flow Control

Matlab, like all programming languages, is designed to save humans from computational tedium. On the other hand, some of these commands might entail performing a sequence of operation to each element of a vector; or again it might be necessary to perform different operations based on the sign of a number. Every programming language contains certain commands designed to alter the flow of execution; either to perform a sequence of commands for each element from a list of parameters, or to change actions based on a condition. Collectively these commands are called “flow control”.

5.6.1 Conditionals

In all flow control we need to make decisions: to continue or not; to take one action or another. In other words we must constantly perform logical tests.

Table 5.1: Matlab Logical Operators

Binary Operator	**Meaning**
==	Is equal to
$<$	Is less than
$>$	Is greater than
$<$=	Is less than
$>$=	Is greater than or equal to
˜=	Is not equal to
&&	And
||	Or

These expressions have boolean values; i.e. they evaluate to either “true” or “false”. In Matlab, “true” is represented by an integer 1, and “false” by 0. Thus, one might examine an expression as follows.

```
>> K = -1;

>> K < 0

ans =

    1

>> K > 0

ans =

    0
```

One must be able to evaluate complicated logical expressions for a number of mathematical tasks. Matlab, like all programming languages, provides a full ensemble of logical operators for that purpose. These are summarized in Table 5.1. One uses parentheses liberally in order to be sure that everything is evaluated properly. Consider the following example.

```
>> a=-1; b=1; c=0;

>> c==0

ans =

    1

>> c==a

ans =

    0

>> a < b

ans =
    1
>> a >=c
ans =
    0
>> a < b && a <=c
ans =
    0
>> a < b || a <= c
ans =
    1
>> a < b || (a<=c && b<c)
ans =
    1
>> (a<b || a>=c) && b<c
ans =
    0
```

Certain flow control structures have become standard for all computer languages, and Matlab supports these.

5.6.2 for

If you want to perform some operation a specific number of times then you should use a "for" construction. In Matlab, the `for` loop has the form

```
for variable=list
    Put commands to be repeated here
end
```

Here is an example that computes the factorial fact of a number K.

```
fact = 1;
for i=2:K
  fact = fact*i;
end
```

This kind of construction is standard. Some variable is "initialized"; a value is put into it to start the sequential calculations. Then the `for` loop is started. The line that does this specifies a parameter name, and the collection of values it will be chosen from. In the example above, the parameter `i` will have a value of 2 at the beginning of the `for` loop. Then the variable `fact` will be assigned the number given by its old value multiplied by 2. The loop continues by moving to the next value in the list – in this case 3. Thus, if `K = 5` then we see that the above example would be exactly equivalent to the Matlab commands below.

```
fact= 1;
fact = 1*2;
fact = 2*3;
fact = 6*4;
fact = 24*5;
```

5.6.3 while

If you need to perform some operation an unknown number of times, until some condition is satisfied, then you should use a "while" construction. It is worth stressing the difference in usage between this and "for". Use a "for" loop when you know ahead of time exactly the set of parameters over which you will iterate; but use a "while" loop when you need to do it until some criterion becomes false.

```
while condition is true
    Put commands to be repeated here
end
```

Here is an example in which we find the largest power n of 2 that is less than or equal to a given positive number K.

```
n = 0;
while 2^n <= K
  n = n+1;
end
% 2^n is now greater than K, so reduce it.
n = n-1;
```

As with the `for` loop, we initialize the parameter n (to 0). Then we test a criterion: if $2^n \leq K$ then we continue by incrementing n. We go around and test again: is $2^n \leq K$? If so, we continue. Indeed we continue until $2^n > K$, after which n is too large, so we reduce it by one. Thus we see that if $K = 13$ then the value we want for n is 3, inasmuch as $2^3 \leq 13$ and $2^4 > 13$. This loop would be exactly equivalent to the following Matlab statements.

```
n = 0;
n = 1;
n = 2;
n = 3;
n = 4;
n = 3;
```

Notice the major difference between this “while” construction and the “for” construction we saw earlier. In the “for” loop, we iterate over a list of parameters that we know beforehand – no decisions need to be made. By contrast, in the “while” loop we do not know beforehand how many times we will iterate, so we must check whether we are finished after each iteration.

5.6.4 if

The “if” construction is not for repeated operations at all. Instead, it is simply a way to have your program make decisions. If some condition is true, then do one set of things; if not, then perhaps do something else, or skip the actions.

```
if condition is true
    Put commands to be performed once here
elseif different condition is true
    Put commands to be performed once here
else
    Put commands to be performed once here
end
```

The `else` and `elseif` statements are optional. If it suits your code, the construction could be as simple as `if condition commands; end`.

The factorial example from the “for” section is pretty stupid if the user puts in a $K < 0$. Here is an “if” construction to give an error if K is negative.

```
if K<0
  error('K must be non-negative.')
end;
```

Observe that in this case we did not use any `else` case at all.

5.7 Matlab Programming

Matlab is at its best when used as a programming language. Yes, it’s nice to be able to throw in a couple of matrices and see what happens, but typically we want to use Matlab to solve more complicated problems, involving several methodical steps of calculations. For such problems, it is best to write a script or a function - that is: a program.

In its simplest form, a script is simply a list of Matlab commands that should be executed sequentially. For example, a simple script to plot a sine function follows:

```
x=-pi:.1:pi;
y=sin(x);
plot(x,y)
```

Table 5.2: Scripts vs. Functions

Script	**Function**
Has access to and uses all variables from the Matlab session.	Has access to only the variables passed as arguments.
Gives the Matlab session access to all variables it creates.	Gives the Matlab session access to only the variables it returns.
Just a sequence of ordinary Matlab commands.	Contains one extra line describing input and output.

If you intended to run this often (not that you would) you could use a text editor (such as gedit) to type these commands into a file called "plotsine.m". After that, inside Matlab, you would need only to type

```
> plotsine
```

to see the plot of the sine function.

It is worth emphasizing that Matlab calls its programs by file name. Whatever the name of the file, it must end in ".m" – that is how Matlab recognizes it. To run the program, you type the name *without* the ".m" in the command line. This is true for both functions and scripts.

More often, scripts are used to automate and abbreviate repetitive work. For that reason we typically use the flow control statements discussed in the previous section. Indeed, flow control statements appear almost exclusively in Matlab scripts and functions.

Sometimes we want to use a script as part of a larger computation – we want it to manipulate a couple of arguments and return a value that we will use for further work. When that is the aim, we usually write a *function* instead of a simple script. The differences between Matlab functions and scripts are summarized in Table 5.2. Basically, a script is just a shortcut for a Matlab session, saving us some typing. A function is a self-contained program that can be reused from one session to another, or even called from other scripts and functions. All scripts and functions for Matlab have file names that end in ".m".

Since scripts and functions are really quite similar, we need only one line to change a script into a function. The line tells what variables in the script are to be passed as arguments, and what variables are to be returned to the calling program. In particular, to make a script into a function, a line such as the following must be added to the top of the script.

```
function [return1,return2,return3]=functionname(arg1,arg2)
```

You may have any number of arguments, and any number of returned variables. The returned variables must all have the same dimension. The function name can be anything you pick except a name that has already been used by Matlab – do not try to name your function "sin".

For example, consider a function that takes two vectors and computes their dot product.

```
function prod=innerproduct(v1,v2)
if size(v1)==size(v2),
  v1 = v1(:);
  v2 = v2(:);
  prod = v1'*v2;
else
  error('The vector inputs must be the same size');
end
end \% ends the function
```

Note that this must be saved in a file called “innerproduct.m”. The name of the file in which a function is stored should always be the same as the name of the function. Since the vectors `v1` and `v2` are not returned from the function, we can change them in any way we like inside the function without worrying that we will mess up work in the calling program. C programmers would say that the arguments are “passed by value”. Thus we feel free to be sure that both vectors are column vectors before we try to compute their inner product.

We would call this function from a Matlab session or another Matlab script simply by creating vectors by any name but with the same dimension, and then calling `innerproduct()`. For example, to compute the mutual dot products of three vectors, consider the following.

```
>> x=ones(1,10);

>> y=1:10;

>> z=22:2:40;

>> p1 = innerproduct(x,y)

p1 =

    55

>> p2 = innerproduct(x,z)

p2 =

    310

>> p3 = innerproduct(y,z)

p3 =

    1870
```

The best way to learn Matlab programming is to examine a collection of examples.

Example 5.3 Suppose we need a function to evaluate

$$\sum_{n=0}^{N}(-1)^n x^n$$

for any x and some choice of N. We could write a Matlab function to do this as follows.

```
function result=powersum(x,N)
result = 1;
for i=1:N
    result = result + (-1)^i * x^i;
end
end
```

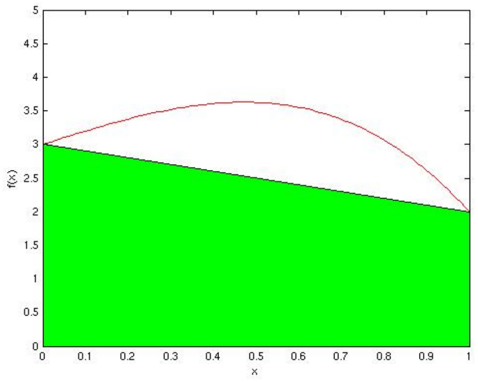
There are a few things to note about this code. First, observe that we need to initialize the result, so we start by putting $(-1)^0 x^0$ into the `result` variable. After that, we just compute each term and add it into the result. Observe that if the user were foolish enough to put a negative value for N as an argument, then the code would still run, but would not give a meaningful result. This is because specifying the range in this format supposes that the increment in `i` is one, but if $N < 0$ the termination of the range is reached immediately.

A second thing to note about this code is that computing powers of numbers is comparatively time consuming compared to e.g. simple multiplication. This is because computing a power of a number usually requires using exponentials and logs. We could alleviate this somewhat by replacing `(-1)^i * x^i` by `(-x)^i`, but even this still requires computing a power of a number. For such a small code it probably does not matter, but we could make this code more efficient by replacing the power by a multiplication.

```
function result=powersum(x,N)
result = 1;
xpower = -x;
for i=1:N
    result = result + xpower;
    xpower = xpower*(-x)
end
end
```

This requires one extra variable, but it actually runs roughly ten times faster than the first code. The point is that when computing integer powers of a quantity, you will save a good deal of time doing that using products, instead of exponential notation.

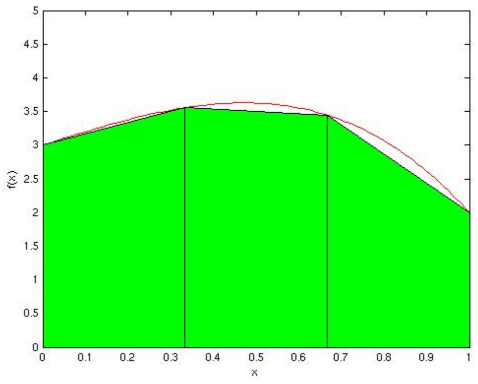
Example 5.4 Suppose we have some function whose integral we want to calculate. One of the simplest ways to do that is to use the composite trapezoidal rule. The idea is to approximate the integral of a function over an interval $[a, b]$ by the integral of the line joining the points $(a, f(a))$ and $(b, f(b))$. This is illustrated in the figure below.



We can thus see that an approximation for $\int_a^b f(x)\,dx$ is $(b-a)(f(a)+f(b))/2$. We could divide the interval into smaller subintervals Choose some positive integer n. Letting $h = (b-a)/n$ and $x_i = a+(i-1)h$ for $i = 0, 1, \ldots, n$ evidently

$$\int_a^b f(x)\,dx = \sum_{i=1}^{n} \int_{x_{i-1}}^{x_i} f(x)\,dx \approx \sum_{i=1}^{n} \frac{h}{2}(f(x_{i-1})+f(x_i)) = \frac{h}{2}\left[f(a)+f(b)+\sum_{i=1}^{n-1} 2f(x_i)\right]$$

This is the composite trapezoidal rule. The area found by this approximation to the integral is illustrated in the next figure.



We can make a function to evaluate an integral using the composite trapezoidal rule as follows.

```
function integral=trapezoid(f,a,b,n)
h = (b-a)/n;
integral = f(a)+f(b);
for i=1:n-1
    integral = integral + 2*f(a+i*h);
end
integral = integral*h/2;
end
```

Note in particular that we save the multiplication by $h/2$ for the last step, *outside* the "for" loop. We would not want to do the extra multiplication and division inside the loop - that would amount to too many extra computations.

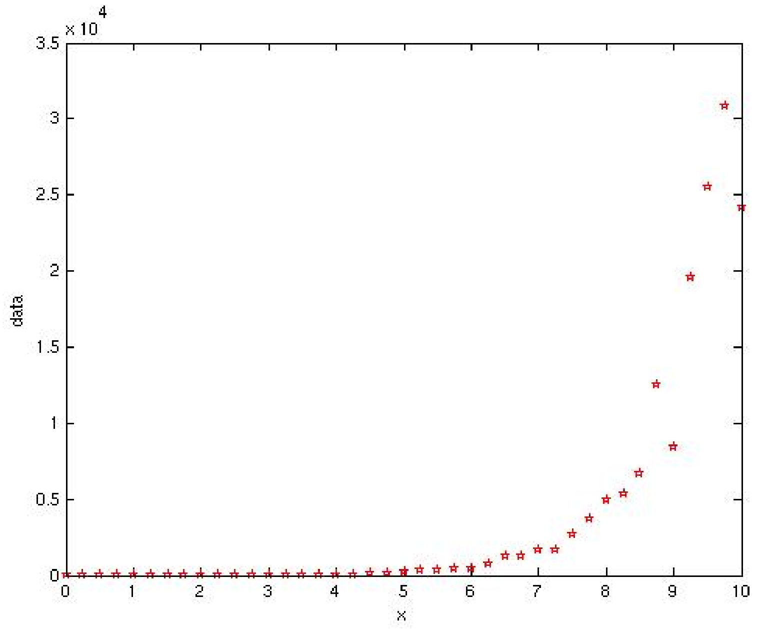
5.8 Advanced Matlab Plots

Matlab is extraordinarily powerful in creating graphics, which can be saved in most of the common formats. Indeed, often computational scientists do their computations using a compiled language such as C, but still use Matlab to plot the results.

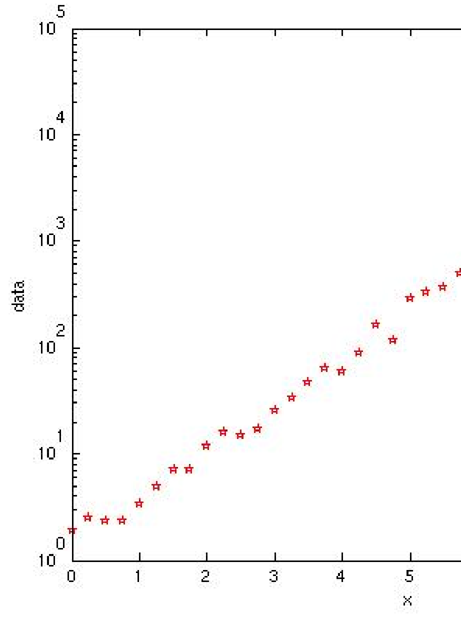
There are basically three ways in which Matlab can produce more sophisticated plots: first, it is possible to superimpose one type of plot on another; there are many types of plotting commands; and finally there is a vast panoply of parameters that can be modified to enhance plots. We already discussed using the `hold` command to superimpose different types of plots. In this section we will discuss some other plot commands, and setting plot parameters.

5.8.1 Plot Commands

The most obvious way to make more complicated plots is to use different functions. Many of these work in very similar ways. For example, in two dimensional on Cartesian oriented axes, the arguments always denote abscissæ, ordinates, and format strings, respectively. We can leave off any particular argument with varying results. The choices for these plots include `semilogx`, which features log scaling on the x axis; `semilogy`, which uses log scaling on the vertical axis; and `loglog`, which obviously uses log scaling on both axes. We like log scaling any time the values in that direction vary as multiples of one another, instead of incrementally. For example, if the x coordinates of our data series increase more or less additively, as with $\{2, 4, 8, 10, 16, 18, 20\}$, then we should use ordinary scaling on that axis. On the other hand, if those x coordinates increase multiplicatively, as in $\{2, 4, 8, 115, 31, 68, 120\}$, then we might want to use log scaling on that axis. The idea is to get as near to a uniform spacing for the points as we can.



(a) Ordinary axis scaling



(b) Log axis scali

Suppose that we want to plot some data that increase roughly exponentially on the interval[0,10]. The result would look as shown in Figure 5.3a. Observe that the curve spends much of its time smashed down on the x axis, showing none of the detail of the noise in the data. In the hope of discovering interesting details in the part of the curve that has small magnitude, we might use a `semilogy` style of plot instead, as shown in Figure 5.3b. Note that now one can see variations in the data, even in the part of the curve that is small in magnitude. Thus, we see that using log scaling for one axis or the other can allow us to see details in function values that vary widely in magnitude over their domains.

Matlab also provides a number of functions for displaying three-dimensional graphics. Typical of these is the `surf` function. This, like all of them, takes a variable number of arguments. In its basic form it plots a matrix of numbers as if they were the values of a function $f(i, j)$, where i and j are the integer indices of the location of the number in the array.

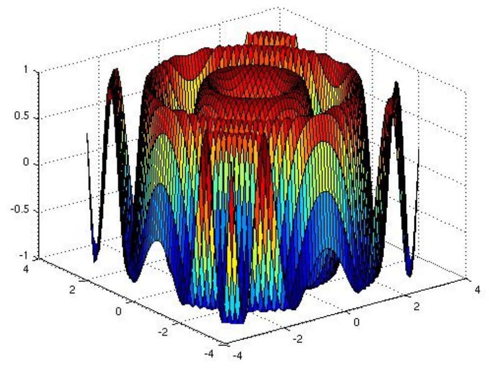
With three arguments, `surf` assumes that the first two are x and y coordinates of the corresponding element of the array being plotted. This is all best seen in an example.

Example 5.5 Suppose that we need to make a surface plot of the function

$\sin(x^2 + y^2)$. We first make matrices of x and y coordinates for the points of the plane over which we want to plot the surface. Then we just use the `surf` function.

```
>> x = -pi:.1:pi;
>> x = x'*ones(1,length(x));
>> y = x';
>> surf(x,y,sin(x.^2+y.^2))
```

The result is shown in the figure below.



There are a number of other functions for plotting scalar functions of two variables, including `mesh`, `contour`, `surfc`, `meshc`. The action of these functions is described in the Plot Commands reference section.

5.8.2 Plot Parameters

Once you have made a plot that is scaled well, with appropriate lines and markers, you might need to customize it further. This can be done in Matlab through the plot GUI, however doing it this way is manual, and if your plot should change or if you need to run it again later, you will have to make all the appearance changes manually again. It is possible instead to program the appearance changes in a way that can be scripted and automatically reproduced.

There are three kinds of parameters that control the appearance and behavior of Matlab figures. First, the figure itself has some properties. These usually pertain to features of the GUI, and we shall ignore them for now. Inside a figure, there will be one or more axes. The word axis is not generic here: it refers specifically to a plot region within a figure. The axis properties control things such as the fonts used in tick markings and labels, the intervals between tick marks, and the axis limits themselves. Finally, each data series displayed

Reference 5.3

Plot Commands

In each of the descriptions below, italicized arguments are optional. The word "format" refers to a format string

plot(x,y,*format*) Basic two-dimensional plot. `x` is a vector of abscissæ, `y` is a vector of ordinates.

semilogx(x,y*format*) This makes the coordinate display on the horizontal axis use a log scale.

semilogy(x,y,*format*) This makes the coordinate display on the vertical axis use a log scale.

loglog(x,y,*format*) This makes both coordinate axes use a log scale.

polar(r,theta,*format*) This makes a polar plot.

surf(x,y,z) This produces a surface plot of values taken from a matrix `z`. Both `x` and `y` must be matrices of the same dimension as `z`, giving x and y coordinates, respectively, for each entry in `z`. If one gives the command as `surf(z)`, then the coordinates from the matrix `z` are plotted against integers giving the indices of the elements plotted.

mesh(x,y,z) This works similarly to `surf`, but makes a wire mesh plot of a surface.

contour(x,y,z) This works similarly to `surf`, but makes a contour plot.

surfc(x,y,z) This works similarly to `surf`, but makes a surface plot with a contour plot in the $x-y$ plane.

meshc(x,y,z) This works similarly to `surf`, but makes a wire mesh plot with a contour plot in the $x-y$ plane.

in the plot has its own settings. These control such attributes as color, line style and thickness, as well as marker style and size.

The simplest way to change plot parameters is through the plot command line. We can provide further options-value pairs of arguments to any plot command to change the appearance of the plot. For example, we could make the line thickness greater in a simple plot as follows.

```
>> x=-pi:.1:pi;
>> plot(x,sin(x),'LineWidth',2)
```

Note that instead of a format string we have a pair of supplemental arguments in the form *name, value*, which give the name of the property to be altered from a default, and the value it is to be altered to. We could have left the format string in just as easily.

```
>> plot(x,sin(x),'r--','LineWidth',2)
```

By default, the parameters in the actual plot command refer to all data series named in the command. Thus, the line width of both curves in the following plot will be 3 pixels.

```
>> plot(x,sin(x),'r--',x,cos(x),'LineWidth',3)
```

On the other hand, this means that we cannot change attributes that have to do with a set of axes this way, and we cannot change attributes for the data series independently, aside from what we can do with a format string.

If we do need to set attributes for data series separately, say set one data series to be displayed with a line width of 2, and the other with a line width of 3, we can get a handle to all the data series a plot command creates as the result of evaluating the plot command. In the following code, we create a plot with two data series. Initially we set the line width of both to 3, but in a later command we change the width of the second one to 2. We also want to change the second curve from one of the rather limited palette that Matlab provides by default, to a color we specify manually.

```
>> p = plot(x,sin(x),'r--',x,cos(x),'LineWidth',3);
>> set(p(2),'LineWidth',2,'Color',[1 .6 0]);
```

As you can guess, `p(2)` is a handle to the second data series – the cosine – whose attributes we change with the `set` function. The color specification is an RGB triple. Each entry in the RGB setting is a floating point number in the interval [0,1]. In this case, the color is a mix of red and green, with somewhat more red than green: it is orange.

If we need to change the attributes associated with the current plot axes, we may use the `gca` variable in the `set` function. The name `gca` stands for *get current axes*, and it always contains a handle to the current plot region. In the previous code, we could additionally change the font size to 16 points by adding another command.

```
>> set(gca,'FontSize',16)
```

Reference 5.4

Plot Attributes

Data Series Attributes

Color The color of the line segments joining the points in the data series. Its value can be either a string name for a stock color, e.g. 'red' or 'r', or it can be an RGB triplet of numbers in [0,1]. For example, [.5 .5 .5] would specify a grey curve.

LineWidth This is the width of the line segments joining the data points in pixels. Its value is an integer.

MarkerEdgeColor The color of the boundary lines of the markers that show where the data points lie. This is specified as with the 'Color' attribute.

MarkerFaceColor The color of the interior of the markers that show where the data points lie. This is specified as with the 'Color' attribute.

MarkerSize The size in pixels of the markers that show where the data points are.

Axis Attributes

FontSize The font size for axis labels, tick mark labels, and so on. The default is 10.

XLim The left and right endpoints of the x axis shown.

YLim The left and right endpoints of the x axis shown.

XTick This allows you to set a vector of values where the x axis tick marks should appear.

YTick This allows you to set a vector of values where the y axis tick marks should appear.

5.9 Matlab Symbolics

Matlab was designed from the beginning as a numerical program. It excels at everything related to numerical linear systems. If this were all it did, it would be an extremely popular piece of software. However, it also has a certain symbolic capability. This is a secondary use of the program, and if your needs are for intense symbolic calculations, then you should probably use different software, but if you have a couple of symbolic expressions to work through that lead to powerful numerical results, then Matlab is ready to help. In this section we will look at some of the simplest features of Matlab's symbolic function array.

The basic idea of symbolic expressions is that they are character strings that describe mathematical expressions. These character strings are interpreted by Matlab so that they can be manipulated algebraically, differentiated or integrated, or used as formulæ for numerical calculations. Matlab provides a standard array of tools to do this.

The simplest symbolic expressions in Matlab are provided in the `inline` function. This allows us to create a function on the fly that evaluates simple symbolic expressions. It takes one obligatory argument – a character string giving the symbolic expression to be evaluated – and an optional argument in which you can specify which character objects are to be interpreted as function arguments. For example,

```
>> f=inline('1+2*x+x.*x');
>> f(0)

ans =

     1

>> f(-1)

ans =

     0
```

defines a function of the single variable x. As you can see, we can then use the new function as if it were any other Matlab object. Matlab figures out that x is the object that should be substituted for. By default, it places these arguments according to the order in which it finds them in the expression. Thus

```
 f=inline('1+a*x+x.*x')
```

makes a function f of two variables a and x, but a is first in the argument list of f; i.e. $f = f(a, x)$. If you want to use the function as $f(x, a)$, then you must define it as

```
 f=inline('1+a*x+x.*x','x','a')
```

For more sophisticated symbolic manipulation, we need to define certain variables as symbolic, instead of numeric. It is easiest to use the `syms` function for this.

```
syms x y
```

This expression warns Matlab that x and y are to be treated as purely symbolic variables, and will not take on numeric values. We can then use those variables in expressions that do not require quotes.

```
>> f=inline(1+2*x+x.*x)

f =

     Inline function:
     f(x) = x.*2.0+x.^2+1.0

>> f(2)

ans =

     9
```

At this point, we can perform algebraic operations on $f(x)$, or on other expressions involving x and y. For example, we could factor $f(x)$.

```
>> factor(f(y))

ans =

(y + 1)^2
```

Since we used the symbolic variable y instead of x in the expression, we get the answer in terms of y. Note that the inline function could not have given us this symbolic answer if we had not defined both x and y as symbolic variables. Observe that the original expression for the action of f used elementwise multiplication, to preserve the ability to use the function on each element of a vector. The factorization of it did not preserve that. Note also that the `factor` function has multiple definitions. If its argument is an integer, then it finds the prime factorization that number.

```
>> factor(42)

ans =

     2     3     7
```

If we try to use factor on a character string in hope that it will perform a symbolic factorization, we will be disappointed.

```
>> factor('x*x+2*x+1')
Error using factor (line 21)
N must be a scalar.
```

An alternative way to create a symbolic expression uses the `sym` function. In spite of having a name very near to `syms`, the `sym` function behaves differently: it converts a character string to a symbolic expression. For example, we could create a single symbolic variable this way.

```
>> z=sym('z')

z =

z

>> factor(z^2-1)

ans =

(z - 1)*(z + 1)
```

Note that the function `sym` returns a symbolic expression – it does not actually defined or change the nature of anything in the character string that is its argument. On the other hand, it is typically used to create an expression that we want to work on.

```
>> g=sym('(t^2-1)/(t+1)')

g =

(t^2 - 1)/(t + 1)

>> simplify(g)

ans =

t - 1
```

Once we have created symbolic variables with `syms`, we have a number of algebraic and calculus operations available to us. In the example above we applied the simplify function, which attempts to simplify the expression by canceling common factors, applying trigonometric or exponential identities, and so on.

```
simplify(sin(exp(x)*exp(-2*x)*exp(1))^2+cos(exp(x)*exp(-2*x)*exp(1))^2)

ans =

1
```

We can rearrange expressions, especially polynomials, by expanding them.

```
>> expand((y+1)^2)

ans =

y^2 + 2*y + 1
```

Or again, we can collect terms involving a certain expression.

```
>> ez=(x^2+4*x+3)*(y^2-5*y+6);
>> expand(ez)

ans =

x^2*y^2 - 5*x^2*y + 6*x^2 + 4*x*y^2 - 20*x*y + 24*x + 3*y^2 - 15*y + 18

>> collect(ez,x)

ans =

(y^2 - 5*y + 6)*x^2 + (4*y^2 - 20*y + 24)*x + 3*y^2 - 15*y + 18

>> collect(ez,y)

ans =

(x^2 + 4*x + 3)*y^2 + (- 5*x^2 - 20*x - 15)*y + 6*x^2 + 24*x + 18
```

Matlab can solve algebraic equations. In the above example, we can see the values of x that would cause `ez` to be zero. So can Matlab.

```
>> solve(ez==0,x)

ans =

 -3
 -1
```

```
>> simplify(solve(ez==1,x))

ans =

   ((y - 2)*(y - 3)*(y^2 - 5*y + 7))^(1/2)/((y - 2)*(y - 3)) - 2
 - ((y - 2)*(y - 3)*(y^2 - 5*y + 7))^(1/2)/((y - 2)*(y - 3)) - 2
```

When we solve for x in a less trivial case, Matlab is still able to solve the equation in terms of y.

Matlab can do summations, whether finite or infinite, using the `symsum` function. This can anywhere from one to four arguments: the first is the summand, the second an index over which the sum is to take place, while the third and fourth represent the lower and upper bounds of the sum, respectively. Incredibly, if one leaves out the index variable argument but includes the lower and upper bounds, Matlab can figure that out and handle it.

```
>> syms i
>> symsum(0.8^i)

ans =

-5*(4/5)^i

>> symsum(r^i,i)

ans =

piecewise([r == 1, i], [r ~= 1, r^i/(r - 1)])

>> symsum(r^i,i,1,Inf)

ans =

piecewise([1 <= r, Inf], [abs(r) < 1, - 1/(r - 1) - 1])
```

Matlab can do various tasks from Calculus, including differentiation, integration, evaluating limits, creating and manipulating power series, and others. Using the function f defined earlier in this section, we find that Matlab can handle indefinite integrals.

```
>> int(f(x))

ans =

(x*(x^2 + 3*x + 3))/3
```

We should probably simplify that answer.

```
>> g=expand(ans)

g =

x^3/3 + x^2 + x
```

We differentiate the result, and expand to make it look like the original form of f.

```
expand(diff(g))

ans =

x^2 + 2*x + 1
```

If a definite integral were required, we could make that happen easily.

```
expand(diff(g))

ans =

x^2 + 2*x
```

In every case when we are dealing with sybolic objects, Matlab seeks an analytic or symbolic solutions. The point is that unlike its ordinary operation, it is not doing floating point computations. Instead, it seeks what we might think of as exact solutions, or none at all. In general we use these capabilities as ways of getting to formulas that we can use for numerical computations.

Reference 5.5

Matlab Symbolics

We provide only the simplest description of the following functions. For details, use Matlab's help utility.

collect This function collects the coefficients for a symbolic variable or expression you specify, grouping terms according to powers of that expression.

diff A function to find the derivative of a symbolic expression.

expand This function carries out algebraic operations on a symbolic expression, distributing and generally expanding products.

inline This defines a function on the fly. It takes a string argument giving an expression that can be evaluated. Matlab determines what the variable is in the expression, or you can tell it explicitly.

int A function to compute the antiderivative or integral of a symbolic expression.

limit Computes the limit of an expression analytically.

numden This returns both the numerator and denominator of a symbolic expression. It should be called as e.g. `[num,den]=numden(x/y)`

solve This solves a system of algebraic equations.

subs This allows us to substitute values for a symbolic variable in an expression.

simplify This tries to algebraically simplify an expression.

sym This allows us to assign a symbolic expression to a variable.

syms Gives Matlab a list of variables that will be considered to be symbolic, rather than numerical.

symsum This attempts to evaluate a symbolically specified sum or series exactly.

taylor This computes a Taylor series for the expression provided as an argument.

Chapter 6

Maple

Maple is a program developed originally at the University of Waterloo, and later by Waterloo Maple Inc. Its original purpose was to do symbolic mathematics, but as with all such programs, it quickly branched out into graphics and related areas. We discuss it here as an example of a broad class of programs that can manipulated symbolic expressions from mathematics.

The first generally-used symbolic manipulation program was Macsyma, created at the Massachusetts Institute of Technology

6.1 Basic Maple

Maple was written for, and possibly by mathematicians. For that reason, its syntax is very like that of mathematics, and is frequently unlike other programming languages. That said, it still has very much in common with other programming languages in terms of flow control, arrays, and other such standard features.

6.1.1 Starting Maple

You may want to change the Maple interface somewhat. Initially Maple forces you to use“2D Math Notation”. Some people like this, but others find this to be a bit painful – it is not always an intuitive interface. If you want to use a more classical command line interface, you can go to Tools Options, and click the “Display” tab. Change input display to “Maple Notation”. Then click the “Interface” tab and change Default format for new worksheets to “Worksheet”. Now when you open a new worksheet you will get the classic Maple input. In what follows we will suppose that you are using command-line input, but that the output is in “2D” mode.

6.1.2 Expressions

One difference between Maple and other languages is the syntax used for assignments. In particular, to assign a value to a variable, one uses the symbol ":=" instead of the simple equals sign. This is actually a common mathematical way of expressing "is defined to be", so it is more consistent mathematically than the equals sign by itself.

```
a := 3: b:= x^2;
```

$$x^2$$

A second thing to note is that every Maple input line should end in a semicolon or a colon. The semicolon tells where the end of the input is, the colon not only ends a command, but also suppresses the output. In "2D" input, the semicolon is optional for command cells that have only one executable line, but you can reduce confusion by continuing to end every line in some kind of colon. In the example above, Maple will echo the value of `b`, but not `a`. Note that in the "2D Math Notation" input format, you do not need the semicolons – it is more line oriented. Any time you press the <`Enter`> key, Maple will process the line you typed in. If you want to make a new line without having Maple process the existing input (i.e. you want a command to go on for several lines), you can get a new line by holding the <`Shift`> key and then hitting <`Enter`>. However, if you want to include several Maple commands in a single execution cell, you must separate them by semicolons or colons. It is best just to get into the habit of placing those at the end of every line.

Apart from those issues, entering expressions in Maple is very like typing them in any language. One major difference is that it is perfectly acceptable to enter expressions in terms of variables that have not been defined. Maple assumes that any variable that has not been defined is to be treated as a symbolic variable.

```
x:=2*a+5;
```

$$11$$

```
y:=cos(x/3);
```

$$\cos\left(\frac{11}{3}\right)$$

```
z:=exp(t^2);
```

$$e^{t^2}$$

6.1.3 Floating point versus exact arithmetic

It is worth noting that Maple assumes that every calculation should take place analytically, unless it is told otherwise. In other words, Maple does not use floating point approximations unless it is told, or at least given an hint. For example, in the last computation from the previous section Maple was asked to evaluate $\cos(11/3)$. It could not do that exactly, so it left the answer in the form $\cos(11/3)$ – the exact answer. If we wanted a floating point approximation to that, we could get it in either of two ways.

```
y:=evalf(cos(x/3));
```

$$-.8652868428$$

```
y:=cos(x/3.0);
```

$$-.8652868433$$

In other words, if Maple sees one floating point number in an expression, it performs all subsequent calculations using floating point arithmetic. If you need to convert an exact expression to floating point, you can use the `evalf` function. Note that the results Maple gives in the above examples differ. In the first case, Maple computed $\cos(11/3)$ in ten digit floating point arithmetic. In the second example, Maple computed 1/3 in ten-digit floating point arithmetic, then substituted for x in ten-digit float point arithmetic, computed the product, committing an error in the tenth digit, and then took the cosine of that approximation. The point is that the later you apply the floating point approximation, the more accurate your answer is likely to be. On the other hand, the longer you require Maple to use symbolic operations, the more time your calculation will take. This is important enough that it bears repeating: you will always be faced with a question of how long to persist with symbolic calculations. Switching to floating point computations will almost always save time; staying analytic will always improve accuracy.

6.1.4 Evaluation

There are issues in evaluating expressions that have nothing to do with getting numerical values. Every time Maple encounters an expression it makes an effort to evaluate that expression in view of the values of variables that it currently knows.

```
a:=b:
a+b;
```

$$2b$$

When Maple encounters the expression `a+b`, it immediately attempts to evaluate the expression in terms of the variables it knows, so it substitutes b for a and produces $2b$ instead of the literal $a + b$. Most of the time this is what we want, but every so often we need to prevent Maple from attempting any evaluation until the opportune moment, namely that moment when numerical values are available for variable substitutions. An example that seems to arise often occurs in plots.

```
f:=t->if t<0 then 0; elif t<1 then 1; else 0; end if:
plot(f(t),t=-1..2)
```

```
Error, (in f) cannot determine if this expression is true or false:  t < 0
```

The point here is that Maple tried to evaluate the expression `f(t)` immediately on entering the function, but doing so required a numerical value for the argument, which was not available until Maple actually starts plotting points. To deal with this, we must delay the evaluation of the expression `f(t)`. We do this by enclosing it in single quotes. The single quote characters prevent evaluation, or rather, the process of evaluating any object in single quotes is to strip off a layer of single quotes. Thus

```
f:=t->if t<0 then 0; elif t<1 then 1; else 0; end if:
plot('f(t)',t=-1..2)
```

gives the plot of the function as required. When Maple first starts the plot, it tries to evaluate the first argument to the plot command, but that just removes the quotes. Maple then proceeds to substitute numerical values into the expression to plot points, but now that numbers are involved, all the comparisons are possible.

Another use of this so-called "unevaluation" is to clear variables. If there is a variable that has been assigned a value, but you want it to be an unassigned symbolic variable again, then you can assign it its unevaluated name.

```
a:=1;
```

$$1$$

```
a:='a';
```

$$a$$

There is a Maple command to perform this same task, but it really offers no advantages. Indeed, one still must be sure when using the `unassign` command to use single quotes on the variable name that is the argument.

```
unassign('a');
```

6.1.5 Functions

Maple makes a strong distinction between expressions and functions. We can define functions in Maple using the -¿ notation.

```
f:=t->t^2;
```

$$t \to t^2$$

```
f(x/3);
```

$$121/9$$

It is very important to note that the notation `f(t):=t^2;` does not define a function. It defines an expression that is put into a variable named `f(t)`. If you want to be able to evaluate a function as e.g. `f(2)`, then you must use the arrow notation.

6.2 Maple Arrays

Maple has a larger than average set of array structures. Moreover, each array type has many ways it can be initialized. These facts combine to make Maple arrays one of the most complicated parts of a complicated language. Here we discuss some of the structures briefly.

6.2.1 Lists

The most basic array structure in Maple is the ordered list. This is a simple indexed list of elements. Each element can have any type, and could even be another list. The elements of a list are not even required to be of the same type. One makes a list simply by typing the elements, separated by commas, enclosed in brackets. For example,

```
mylist := [1, 2, 3];
```

We can then refer to elements of the list using a different bracket notation: mylist[i] refers to the element with index i in mylist. Thus, in the example above, mylist[2] is 2. It is worth emphasizing the two different uses of brackets here: in one case they signify that we are defining a list, in the other, they enclose the index of elements in that list.

6.2.2 Sets

The set differs from the list in Maple in that it is not ordered, and therefore does not contain repeated elements. Sets are created using braces instead of brackets:

```
myset:={1, 2, 3}
```

The following example illustrates several of the differences between lists and sets.

```
a_set:={[1,2],2,2};
```

$$\{2, [1, 2]\}$$

```
a_list:=[[1,2],2,2];
```

$$[[1,2],2,2]$$

Note that the repeated 2 in the definition for `a_set` does not appear in the result, and that the order that Maple echoes after the set is defined differs from the order in which the elements were typed. Thus, while we can refer to elements of the set by index, in this case `a_set[2]` is the list [1,2], even though that was originally typed in the first position. By contrast, the list remains exactly as it was originally typed.

6.2.3 Arrays

The next array structure is actually called an array. Arrays differ from the earlier structures in that they have specific dimensions which need not start at 1. While you can make a list larger just by defining a new element, you must specify the size of an array at the time of definition, and cannot change that size except by a redefinition. There are many ways to define arrays; perhaps their use is best described using an extended example.

```
A1:=Array(1..2,[1,2]);
```

$$\begin{bmatrix}1 & 2\end{bmatrix}$$

```
A2:=Array(1..2,1..2,[[1,2],[3,4]]);
```

$$\begin{bmatrix}1 & 2\\ 3 & 4\end{bmatrix}$$

```
v:=Array(1..2,1..1,[[1],[1]]);
```

$$\begin{bmatrix}1\\ 1\end{bmatrix}$$

```
w:=Array(-1..0,1..1,[[1],[1]]);
```

Array(-1..0,1..1,(-1,1)=1,(0,1)=1,
datatype=anything,
storage=rectangular,
order=Fortran_order)

```
w[-1,1];
```

$$1$$

Note that when the array is defined using negative indices, then the echo does not show its 2-D form.

The truth is that we do not use the Array form that often. It provides more structure than sets or lists, but not so much that it is used commonly.

6.2.4 Vectors and Matrices

Finally, Maple lets us define vectors and matrices, and to operate on those. There are several ways to define these.

```
v := <1,2,3>;
v := Vector(1..3,[1,2,3]);
```

are equivalent ways to make a 3-vector. Likewise

```
m := <<1,2,3>|<4,5,6>>;
m := Matrix(1..3,1..2,[[1,2,3],[4,5,6]]);
```

are equivalent ways to make a 3×2 matrix. Matrices and vectors are basically like Arrays, except in that they are always indexed starting at 1. There are no elements with negative indices for vectors or matrices. Moreover, the notions of matrix and vector operations are clearly defined for these structures, so we can add them and multiply them when their dimensions allow. Using the linalg or LinearAlgebra packages, we can do much more sophisticated things with them also.

```
with(LinearAlgebra):
v := <1,2,3>:
m := <<1,2,3>|<4,5,6>>:
Transpose(m).v;
```

$$\begin{bmatrix} 1 & 4 \\ 3 & 2 \end{bmatrix}$$

```
M :=<m | v>;
```

$$\begin{bmatrix} 1 & 4 & 1 \\ 2 & 5 & 2 \\ 3 & 6 & 3 \end{bmatrix}$$

```
M[2,3]:=-1:
LinearSolve(M,v); [1] [0] [0]
```

$$\begin{bmatrix} 1 \\ 0 \\ 0 \end{bmatrix}$$

Notice that the matrix multiplication symbol is the period '.'. This is to distinguish the matrix operation from the ordinary scalar multiplication.

6.2.5 Manipulating Arrays

Once you have created an appropriate data structure for a job, you must put values into it, change those values, copy them, and manipulate elements and other parts of those structures. Maple uses a variety of notations to do that.

The basic way to refer to an element of almost any array is to use brackets to enclose the index of the element you want to use. This works for lists, arrays, sets, as well as vectors.

6.3 Flow Control

Like all programming languages, Maple has a full complement of flow control statements. We know what statements we need from earlier discussions: we need a *for* construction to deal with sequences of commands for which we know exactly how many times they will be repeated; we need a *while* construction when we repeat sequences of commands until some condition is satisfied; and we need an *if* construction in order to make decisions.

6.3.1 If

The basic syntax for *if* in Maple is

`if` *conditional* `then` *statement sequence*;
`elif` *conditional* `then` *statement sequence*;
`else` *statement sequence*;
`end if`;

As always, the `elif` and `else` are optional.

The conditionals have a similar form to those in other languages. There are a few differences. First, conditionals in Maple do not have a truth value outside of the “if” statement context. In other words, the expression “5¿2” has no truth value in Maple, but `if 5<2 then c:=3; end if;` has meaning. If you want to find the truth value of a conditional outside of an ”if” statement, you must use the `evalb` function. Here is an example.

```
5=3+2;
```

$$5 = 5$$

```
evalb(5=3+2);
```

$$\textit{true}$$

```
if 5>2 then
  x:=3;
elif 5=2 then
  x:=103;
end if;
```

$$3$$

It is worth a special note that Maple uses := for the assignment operator, which frees up = to be used to test equality. This differs from many other languages, which typically use == for that purpose. Table 6.1 summarizes the logical operators.

Table 6.1: Maple logical operators

Symbol	Meaning
=	Is Equal To
<>	Is Not Equal To
>	Is Greater Than
>=	Is Greater Than Or Equal To
<	Is Less Than
<=	Is Less Than Or Equal To
and	Logical And
or	Logical Or

6.3.2 For

The syntax for repetitive loops is a bit different in Maple.

`for` *variable* `from` *expression*
`by` *step* `to` *expression* `do`
statement sequence
`end do;`

You will find this to be fairly intuitive. The `by` clause is optional if the variable is to be incremented by one each time.

```
for i from 1 to 10 do
  x[i] := i:
end do;
y := 0:
for j from x[3] to x[3]^2 do
  y := y+j:
end do;
the_answer_to_life_the_universe_and_everything:=y;
```

$$42$$

Note that the task of the first loop could have been done more easily using `x:=[seq(i,i=1..10)];`

6.3.3 while

Maple also includes a `while` command. As with all such commands, this is used for a repetitive sequence when it is unclear a priori how many times the statements must be repeated. The syntax here is

`while` *conditional* `do`
statement sequence
`end do;`

For example, one could find the power of 1/2 that is smaller than some given number as follows.

```
power:=0;
while 2^(-power)>=number do
  power := power+1;
end do;
```

6.4 Maple Programming

One can write functions in Maple just as in other languages. However, in Maple, the word function refers to mathematics, and defining a function properly uses the syntax described in Section 1.

```
f:=x->2*x+3:
f(2)
```

$$7$$

An actual programming construct that takes arguments and returns an output is called a *procedure* in Maple. The action of a procedure in Maple is ultimately similar to that of a function, but their construction is different, and in some ways they are used differently.

To create a procedure, the basic syntax follows a pattern looking like the following.

procedure_name := `proc(`*argument list*`) local` *variable list*;

`end proc;`

The procedure_name is the name you make up for your procedure. Be sure that it does not conflict with any names that Maple already uses. For example, it would be a bad decision to try to call your procedure `plot`. The argument list is just a list of names for the variables that will be supplied as arguments. The variable list is different from what we have seen in e.g. Matlab: Maple wants us to *declare* variables before we use them. In other words, we should list all the variables that will be used inside the procedure, but will be private to the procedure. As it happens, if we use a variable without declaring it to be local, then Maple will assume that it is local by default. However, when it does so, it will send us a warning message that we probably do not want to see, so it is best to declare our local variables explicitly.

The reason Maple makes such a production out of the declaration of variables is that it also uses global variables. Both local and global variables are available for use inside the procedure. The difference is that the global variable is also available to the Maple session that calls the function where it was declared. Consider the following procedure.

```
setvar:=proc() global one;
one := 1;
end proc;
```

Running this procedure defines a variable called `one` whose value is available not only inside the procedure, but also in the Maple session. In particular,

```
one; setvar: one;
```

gives the result

```
one
 1
```

This difference in the *scope* of a variable is a common feature of most programming languages. Maple just forces you to address it. Typically our functions will require a good number of local variables, but will not require any global variables.

The Maple procedure takes input arguments as normal: a comma separated list of the names of local variables that will be assigned those argument values.

Chapter 7

Python

Python is a computing language whose development started in the late 1980s. The person responsible is Guido Van Rossum, a researcher at the Centrum Wiskunde & Informatica in the Netherlands. Version 2.0 of the language appeared in 2000, at which point it could be said that the language had achieved significant maturity. It has been adopted as a development platform for powerful numerical packages in recent years because of its portability, ubiquity, and the fact that it is free. Python is available for all operating systems and supported by several software development platforms. It makes structured programming easy, and is fairly easy to read.

Though version 2 marked a good level of maturity for the language, Van Rossum decided to rework the language in version 3. This version is not backwards-compatible. Instead, it represents an effort to add functionality, improve structure, and generally clean up the language. Version 3 is the future of the language, but versions 2.6 and 2.7 have been updated with some of the features of version 3, and and support all libraries that have been developed over the years. In this text we will discuss version 2.7.

With the creation of packages such as NumPy, SciPy, SymPy, and Matplotlib, Python is now a mature and powerful extensible environment for the development of mathematical and scientific software, and is used throughout many disciplines. With these additions, Python has come to be very commonly used for interdisciplinary research spread across diverse institutions and labs. It is arguable that it is even more commonly used now than Matlab.

7.1 Getting Python

One of the best things about Python is that it is free. While Matlab and Maple are strongly proprietary, and quite expensive, Python provides most of the same functionality without any expense. The price you pay is perhaps a bit of trouble in installing it. Many distributions make this straightforward, but problems can occur.

In order to use Python for scientific computing, one needs not only the language itself, but also a collection of additional packages. A fairly complete system can be created by installing the following packages.

- Python itself. This can be downloaded from `http://python.org`. There are binary packages available for Windows, Macintosh, and various distributions of Linux, in both 32- and 64-bit architectures.
- NumPy, SciPy, and Matplotlib. These can be downloaded together or separately, depending on the distribution. There are details concerning how and where to get them at `http://www.scipy.org/install.html`.
- SymPy. There are other alternatives for symbolic manipulation packages that add on to Python, but this package seems to be developing quickly. It can be found at `http://code.google.com/p/sympy/downloads/list`.

There are various ways to run Python. It can be run in a basic command line, or a slightly more advanced command line called iPython. It can also be run in a batch mode – that is using scripts that start their own shells and run possibly independently of the process that started them. In our discussion, we will usually assume we are in the simplest command line, or else we will write scripts that can run in a command line or in a file we read into Python.

7.2 Python Basics

7.2.1 Python as Calculator

The simplest way to use Python is as a command line. In whatever operating system you use, open a "terminal" or a "command line", and type "python". You should see a prompt comprising three greater-than signs: "$>>>$". This is where you can type Python commands interactively. You could immediately use this as a simple calculator:

```
>>> 3+4*498.52
1997.08
```

The syntax rules for entering simple arithmetic expressions are very familiar - the same as for most spreadsheet programs. Note that there is an operator precedence: multiplication and division first, then addition and subtraction. Powers are applied using the `pow()` function, or using ** to indicate exponentiation:

```
>>> pow(3,2)
9
>>> 3**2
9
```

Note in particular that the operator we might be familiar with for exponentiation does *not* work. We cannot use the caret (ˆ) to indicate a power.

One thing that will be unfamiliar to some is that Python distinguishes between integers and floating point numbers. This is not true in e.g. Matlab. When we make Python divide one integer into another, it always truncates the remainder – *the result when we divide integers is always an integer*.

```
>>> 7/8
0
```

This is the source of many errors in Python codes. Be careful. As soon as Python sees one number that is entered with a floating point representation, then it performs the calculation in floating point. Operator precedence is still an issue to be concerned with.

```
>>> 7.0/8
0.875
>>> 7/8.0
0.875
>>> 7./8.
0.875
>>> 7./8 + 7/8
0.875
```

Here we see that as soon as one number is typed with a decimal point, then the entire arithmetic unit becomes a floating point calculation. However, in the last line we know that operator precedence causes the two quotients to be evaluated separately before they are added together. Thus, Python evaluates the first quotient as a floating point calculation and gets 0.875, and it evaluates the second quotient as an integer calculation and gets zero. It then sums the result floating point 0.875 and the integer 0 as a floating point calculation, and gets 0.875. That might not be what we wanted. Force yourself into the habit of typing numbers using a decimal point, unless you really mean to treat them as integers.

As with most programming languages, Python accepts numbers typed in a curious scientific notation.

```
>>> 2.5e4
25000.0
>>> 2.5e-4
0.00025
```

Thus, we might have grown used to writing numbers in scientific notation as e.g. 2.5×10^4, but Python prefers using juxtaposition to denote that product, and it uses the letter “e” to signify that the exponent for 10 is coming up.

You can run commands you already typed anew by using the up-arrow keys. Just press the up-arrow to see the command you used last, then hit enter to run it. Pressing the up-arrow three times gets the line you ran three commands ago. You can even edit the commands using the arrows to move around, backspace to delete characters, adding characters wherever the cursor points.

Table 7.1: Pros and Cons of package loading methods

`from numpy import *`	`import numpy as np`
Uses every member with no prefix	Requires `np.` prefix for all package member functions and variables
Might overwrite an existing Python function	Makes sure every existing function remains available

7.2.2 Importing Packages

All this occurs in Python itself, without any need for the special mathematical packages. However, if you want to do anything genuinely mathematical, you will need to load a package or two. In particular, without help, Python knows nothing about trigonometric functions. This is one of the simplest parts of the Numpy package: it provides a library of standard mathematical functions. To load the Numpy package, type the following.

```
>>> from numpy import *
```

If this is successful, there will be no numbers returned, no messages, no nothing. Python does not brag when it gets something right. It does whine when you type something wrong. Here you can see the general syntax for loading things from packages. We told it to look in the Numpy package, and the asterisk indicates that it should load every function in that package. If we had wanted only one function, we could have just typed the name of that function instead of using the wild card "*".

```
>>> from numpy import pi
```

On the other hand, if we want to import the entire package under the name numpy, we can type only "import numpy".

```
>>> import numpy
>>> numpy.pi
3.141592653589793
```

In this case, Numpy is imported as an instance of a class. Every function and variable in Numpy is addressed using a prefix `numpy.`, where the period indicates that the thing following it is a member of the `numpy` object. We will discuss this aspect of Python much more later.

That seems like a lot of typing – if we want to import Numpy this way, but want a shorter name for our Numpy object, we can use the form

```
>>> import numpy as np
>>> np.pi
3.141592653589793
```

You should wonder why we would want to use one method of loading over another. We have discussed some of the pros and cons for each approach in Table 7.1. In this document we will always import Numpy using the `from numpy import *` form, but we will routinely import Sympy using the other form. This is because both Numpy and Sympy have e.g. `sin()` functions that behave differently, and we will need to have both available depending on whether we want numerical or symbolic results.

7.2.3 Numpy Functions and Variables

Once Numpy has been loaded, we can evaluate e.g. a cosine.

```
>>> from numpy import *
>>> cos(9)
-0.91113026188467694
>>> cos(pi)
-1.0
>>> pi
3.141592653589793
```

Python knows all the usual trigonometric functions, including `sin`, `cos`, and `tan`. It does not know about the secant or cosecant functions; we need to evaluate those as reciprocals of cosines and sines. It does know about inverse trigonometric functions: `arcsin`, `arccos` and `arctan`. The basic hyperbolic functions are defined: `sinh`, `cosh`, and `tanh` – again we construct the others as reciprocals of those given. Numpy provides `log` and `exp` functions. Note that `log` refers to the natural log function. If we want to evaluate a log in base 10 we must use the `log10` function.

Note that Numpy knows the value of π to double precision floating point accuracy. It also knows the value of e.

```
>>> e
2.7182818284590451
>>> e**2
7.3890560989306495
>>> exp(2)
7.3890560989306504
```

Observe that in this case there is one small problem. We know that $\exp(x) = e^x$ by definition. However, when we actually evaluate e^2 and $\exp(2)$ in floating point arithmetic, we get a more accurate number from the exponential function than from squaring e. There is significant roundoff error in the latter calculation. Always use the built-in function if you can. It is more accurate, and it is faster too.

We have seen that `pi` and `e` are predefined variables in the numpy package. We can define any other variable we want using an equals sign.

```
>>> piover4 = pi/4.
>>> piover4
0.7853981633974483
```

This is a standard operation in all programming languages. The equals sign means: "evaluate the expression on the right of the equals sign and put it into a variable associated with the name on the left." After that is done, you can get that value by typing the name of the variable any time. Aside from that, you can use the variable in place of a number in any mathematical operation.

As in all programming languages, it is important to emphasize that the equals sign does not signify mathematical equality. It is perfectly acceptable to make statements in the nature of `a=a/2.0` or `b=b+1`. In a mathematical setting, the first would imply that `a = 0`, and the second would simply be impossible. In the computing context, however, all the first statement means is that we should take the value of `a`, divide it by 2 in a floating point sense, and replace the value of `a` by that number. The second statement means that we take the current value of `b`, add an integer 1 to it, and replace that number into the variable `b`.

Because these operations for modifying variables are so common, Python provides some special operators to perform them. These are the `+=`, `-=`, `*=`, `/=`, and `**=` operators. The statement `a = a/2.0` can be replaced by `a *= 0.5` or by `a /= 2.0`. Likewise `b = b+1` can be typed as `b += 1`. In the same way, the `**=` operator raises the number to its left to the power on its right.

```
>>> a=2
>>> a**=2
>>> a**=2
>>> a**=2
>>> a
256
```

7.2.4 Defining functions

You will often want to define your own functions, to evaluate at various places. Python makes that easy, albeit less than intuitive. To define the function, use the `def` keyword, tell the function name and what arguments you want the function to accept, and then say what you want to do with those arguments. Let's look at a definition.

```
>>> def g(x):
...   return x*x*x
...
>>> g(2)
8
>>> g(2*3)
216
```

There are several details to attend to here. First, note the colon at the end of the line with def in it. This tells Python that there is more function definition coming. When you hit <Enter> then Python types the ellipses (...) for you. These are to help you line up your text properly, the point being that indentations are very important. We put in two leading spaces to indent the contents of our function definition, then told the function to "return" the cube of the argument. When Python typed the next set of ellipses, we just hit <Enter> to end the function definition. Thus, `x` represents the number where we want to evaluate the function, and the result of the evaluation is what appears on the `return` line. After that `g(2)` evaluates to the cube of 2; `g(2*3)` evaluates to the cube of 6.

We cannot emphasize enough that the indentation (or "leading spaces", if you prefer) is very important. Not to use an indentation would be an error; using indentations of different lengths would be an error. Here is a definition of a multiline function.

```
>>> def poly(x):
...   y = 2*x
...      return y*y+x+1
File "", line 3
      return y*y+x+1
      ^
IndentationError: unexpected indent
>>> def poly(x):
...   y = 2*x
...   return y*y+x+1
...
>>> poly(3)
40
```

The first attempt to define the function `poly` failed because we used two spaces as the indentation for the first line inside the function definition, but then used four spaces for the next line. The number of spaces you choose for your indentation does not matter, but once you pick a number, you must use the same sized indentation for every line of the function. We will always use two spaces for our indentations in this text.

7.3 Data Types

One of the things that can make Python a little tricky is the different interpretations it can put on a value. Python uses a few simple data types, and any time a value is entered, Python must decide how that value is to be interpreted. Python can support very sophisticated user-constructed data types, but we will only concern ourselves here with the basic predefined ways that input can be interpreted.

The most important data type for mathematicians and scientists is the floating point number. By default, Python interprets any number that includes a decimal point as a double precision floating point number. We will not discuss the true binary representation of these numbers. In order to make good use of Python, we need only to observe that floating point representations are effectively writing numbers in scientific notation: $a \times 10^b$. The number a is called the mantissa of the number, while b is the exponent. In a double precision representation, the number a may be thought of as a sixteen-digit number between -1 and 1. This is not exactly right, but it is a decent rule of thumb. The number b is an integer between -1022 and 1023, inclusive.

When you type `a=3` at the prompt, Python interprets the number 3 as an integer. This is probably what you want, but it might lead to arithmetic results that surprise you. Consider the following Python commands.

```
>>> a = 3
>>> a/4
0
>>> float(a)/4
0.75
```

We have already seen that when Python does arithmetic with integers, it always truncates the result to an integer. Thus in Python, `3/4=0`. If we want to get floating point answers, at least one number in the computation must be in a floating point representation.

We can convert integers to floating point numbers in Python using the `float` and `double` functions. These are almost the same – two names are provided because in some programming languages there are differences between float and double types. There are no such differences in Python, but note that float is built in to Python, while double comes from Numpy, and hence is slightly different. Interestingly, it turns out that the `double` function is somewhat slower than `float` for scalar arguments.

```
>>> from numpy import *
>>> a=3
>>> 1/a
0
>>> 1/float(a)
0.3333333333333333
>>> 1/double(a)
0.33333333333333331
```

There is one other major difference between `float` and `double`; namely that `double` accepts array arguments. We can only convert scalars to floating point numbers using `float`, but `double` can convert an entire array at once.

By the same token, we can truncate floating point numbers to integers in Python using the `int` command.

```
>>> pi
3.141592653589793
>>> int(pi)
```

```
3
```

Strings (in any programming language) are just plain ol' text. Computers like to deal in numbers, so when we want to type plain ol' text, we must enclose it in quotation marks. We can use any kind of quotation marks we want, whether single or double, just so long as we match them properly.

```
>>> hi="Hello, world!"
>>> bye='Goodbye, cruel world!'
>>> wut="What is this 'world' you speak of?"
>>> hi
'Hello, world!'
>>> bye
'Goodbye, cruel world!'
>>> wut
"What is this 'world' you speak of?"
```

Evidently we can enclose one quote inside another, so long as we manage the types of quotation marks used. There is one other type of quotation mark, created by typing three single-quotes together. The `\n` you see is a representation for a newline character – the invisible character a computer uses to move to the next line. These are actual characters in the strings, which can be seen when we echo those strings on the command line.

```
>>> longstring='''This is a preposterously long string,
... which we use to illustrate that we can be verbose.'''
>>> longstring
'This is a preposterously long string,\nwhich we use to ...'
```

Note that the ellipses at the end of the output are not what appears; they are required because this print format does not permit us to display the entire string. This last example also illustrates that there are, in fact, three different sets of quotes in Python: the single-quote, the double-quote, and the triple-quote, which we make using three single-quote characters. The point here is to be able to put quotes inside quotes inside quotes.

Part of what makes this work is Python's ability to distinguish character data from numerical data; however, one critical pieces is the ability of Python to create long collections of similar data types. These are called lists or arrays. Strings are actually long lists of character data. We will discuss this more later.

7.4 Arrays

Python needs to be able to handle vectors and matrices with sophistication if it is to be truly useful in mathematics, and as it happens, it can. The most basic array structure in Python is called a list, and is simply an index ordered list of objects. We will not discuss this much here, since it is of very little use mathematically. Instead, we'll go directly to the useful array structure of Numpy. We can import Numpy again, then create a couple of arrays and add them together.

```
>>> v=array([1,2,3])
>>> u=array([-1,-2,-3])
>>> u+v
array([0, 0, 0])
```

What happened here? When we imported Numpy, one thing it did was to provide access to several special functions and classes. One of those classes is array, which effectively implements mathematical row vectors. We can do many things with arrays that we would not be able to do with the more basic Python list structure. We can create an array by filling an ordinary Python list (i.e. the contents of []) with numbers, and then passing them to the array class in Numpy. Once the arrays are created, we can add them, subtract them, and do other matrix operations in a way that is sometimes intuitive, or other times uses more functions from Numpy. Observe too what happens when you multiply arrays.

```
>>> v=array([1,2,3])
>>> u=array([-1,-2,-3])
>>> u+v
array([0, 0, 0])
>>> u*v
array([-1, -4, -9])
```

The array class can do structures of higher dimension as well, but it has one minor flaw, from a mathematician's point of view: it does not know about matrix multiplication. It can do that using the `dot` function, so if you like the array class, you may use it. On the other hand, if we plan always to do only mathematics, we can make Python behave in a more natural fashion by using the `matrix` class. This is easiest to see in action.

```
>>> v=matrix([1,2,3])
>>> u=matrix([-1,-2,-3])
>>> v matrix([[1, 2, 3]])
>>> v.T matrix([[1], [2], [3]])
>>> u*v.T matrix([[-14]])
>>> u.T*v matrix([[-1, -2, -3], [-2, -4, -6], [-3, -6, -9]])
```

Here we defined both `u` and `v` as 1×3 matrices. We were then able to transpose `v` and even multiply `u` by the transpose of `v`. There is some notation here that might be unfamiliar to you. We wrote the transpose of `v` as `v.T`. The period does not indicate multiplication, as it might in mathematics. Instead, it is a symptom that `v` is actually a self-contained computing object, and `T` is one function associated with that object. You will see this often in Python. Notice also that the meaning of the asterisk has changed when using matrix objects. When we use an asterisk to multiply ordinary arrays, the result is a new array containing the elementwise product. When we use it on matrix objects, the result is the matrix product.

The choice of whether to use the matrix or the array structure is difficult. The former does give us matrix arithmetic that we are used to in mathematics, but most Numpy functions prefer to work with array structures, and return those when asked. Moreover, the array structure is very useful when dealing with collections of values, such as points at which to plot a function. In these pages we will use both structures, and emphasize that it is important to be versatile.

We can change elements of matrices and arrays easily. We can refer to a single element of an array by giving the index of the element in brackets. In the case of a matrix, we use row and column indices in the brackets. There is one important thing to note about these indices: they start at zero. Thus, in a 3×3 matrix the element in the lower right corner has indices [2,2]. Have a look.

```
>>> w=u.T*v
>>> w[2,2]
-9
>>> w[2,2]=pi
>>> w
matrix([[-1, -2, -3], [-2, -4, -6], [-3, -6, 3]])
```

In the first command, we defined a matrix $w = u^T v$. Then we just asked Python to tell us the value of the entry in the lower right corner of that matrix. We then changed that value to π. Wait – since when is $\pi = 3$?!

It turns out that by default, Python assumes that the entries in a matrix are all integers. When we try to put nonintegers in to the integer matrices, Python rounds them off to integers. This is probably not what we want ordinarily. This leads to something unpleasant: when we make matrices, we must remember to create them using the `double` keyword (which stands for double precision - i.e. 64-bit floating point). For the moment, we can just make a new matrix `W` that is double precision.

```
>>> W=matrix(w,double)
>>> W matrix([[-1., -2., -3.], [-2., -4., -6.], [-3., -6., -9.]])
>>> W[2,2] = pi
>>> W matrix([[-1. , -2. , -3. ], [-2. , -4. , -6. ],
                                       [-3. , -6. , 3.14159265]])
```

Again, we have doctored the output here so it fits within the margins. It is a bit troublesome to have to monitor the data type of a matrix, but it will usually not be a problem – you will ordinarily not be typing matrices yourself. Instead you will be loading them from data or computing the entries in some way. In particular, we typically want to create a matrix an set all of its elements initially to have zero. We can do that by using the `zeros` function.

```
>>> A=zeros([3,3])
>>> A
array([[ 0.,  0.,  0.],
       [ 0.,  0.,  0.],
       [ 0.,  0.,  0.]])
       >>> A=matrix(A)
>>> A=matrix(A)
>>> A
matrix([[ 0.,  0.,  0.],
        [ 0.,  0.,  0.],
        [ 0.,  0.,  0.]])
```

Note that the `zeros` function returns an array structure. If we want to make that into a matrix, we must do it ourselves. That behavior typifies the actions of Numpy – the array structure is almost always preferred. It is probably a good thing to get used to it.

7.5 Ranges

It takes a long time to type in the entries of a matrix of any significant size. We do not have time to do that. Fortunately, Python has sophisticated ways of filling and manipulating arrays and matrices. These are similar to those in Matlab, and like the Matlab notation, the ideas take some getting used to.

If we want to fill a list with an ordered collection of numbers starting with 0 with increments of 1, this is easy: viz.

```
>>> v=range(10)
>>> v
[0, 1, 2, 3, 4, 5, 6, 7, 8, 9]
```

There are several important things to note here. First, we see immediately that asking for range(10) gives us ten numbers, but they start at zero, so they end at nine. Second, observe that the result is not an array, not a matrix; instead it is an ordinary low-level Python list. This is enough to count objects in a "for" loop or something, but we will often want more structure. The range command is native Python - not Numpy. Naturally, Numpy provides a command similar to range. It is called `arange`.

```
>>> from numpy import *
>>> v=arange(10)
>>> v
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

This time the result is a Numpy array. It gives us a bit more structure. The numbers still start at zero. If you actually wanted to start at 1 and go to 10, you would have to tell Python explicitly to do that. The `arange` command can take up to three arguments. If it has two arguments, then the first is interpreted as the starting number, and the second is one greater than the ending number.

```
>>> u=arange(1,11)
>>> u
array([ 1, 2, 3, 4, 5, 6, 7, 8, 9, 10])
```

The `arange` function can be used to fill arrays in other ways as well. For example, if you wanted to create a partition on the interval [-1,1] with uniform spacing of 0.1, that could be done easily by using the third possible argument of arange. That argument specifies the increment. We have seen that by default the increment is one, but if we give that third argument it changes the increment to that value. Note that in this case the second argument becomes one increment greater than the ending number. There is a little bit of room for fudging this. Have a look.

```
>>> x=arange(-1,1.2,0.2)
>>> x
array([ -1.00000000e+00, -8.00000000e-01, -6.00000000e-01,
-4.00000000e-01, -2.00000000e-01, -2.22044605e-16, 2.00000000e-01,
4.00000000e-01, 6.00000000e-01, 8.00000000e-01, 1.00000000e+00])
>>> x=arange(-1,1.1,0.2)
>>> x
array([ -1.00000000e+00, -8.00000000e-01, -6.00000000e-01,
-4.00000000e-01, -2.00000000e-01, -2.22044605e-16, 2.00000000e-01,
4.00000000e-01, 6.00000000e-01, 8.00000000e-01, 1.00000000e+00])
>>> x=arange(-1,1.01,0.2)
>>> x
array([ -1.00000000e+00, -8.00000000e-01, -6.00000000e-01,
-4.00000000e-01, -2.00000000e-01, -2.22044605e-16, 2.00000000e-01,
4.00000000e-01, 6.00000000e-01, 8.00000000e-01, 1.00000000e+00])
>>> x=arange(-1,1,0.2)
>>> x
array([ -1.00000000e+00, -8.00000000e-01, -6.00000000e-01,
-4.00000000e-01, -2.00000000e-01, -2.22044605e-16, 2.00000000e-01,
4.00000000e-01, 6.00000000e-01, 8.00000000e-01])
```

We have modified the Python output somewhat so it fits on the page. What we are looking at here makes the actual algorithm used to generate ranges evident. If there are three arguments s, n, i, then Numpy starts at s and appends elements to the array in steps of i until the number is greater than or equal to n. If there are only two arguments, Numpy assumes that $i = 1$. If there is only one argument Numpy assumes additionally that $s = 0$. Simple...

Along with the ability to create arrays quickly, Python allows changes to entire chunks of lists, arrays, or matrices. In this case we can specify ranges for indices using a notation with a colon. It is easier to see an example first.

```
>>> x=matrix([arange(-1,1.1,0.5)])
>>> A=x.T*x
>>> A
matrix([[ 1. , 0.5 , 0. , -0.5 , -1. ],
[ 0.5 , 0.25, 0. , -0.25, -0.5 ],
[ 0. , 0. , 0. , 0. , 0. ],
[-0.5 , -0.25, 0. , 0.25, 0.5 ],
[-1. , -0.5 , 0. , 0.5 , 1. ]])
>>> A[1:3,1:3]=1
>>> A
matrix([[ 1. , 0.5 , 0. , -0.5 , -1. ],
[ 0.5 , 1. , 1. , -0.25, -0.5 ],
[ 0. , 1. , 1. , 0. , 0. ],
[-0.5 , -0.25, 0. , 0.25, 0.5 ],
[-1. , -0.5 , 0. , 0.5 , 1. ]])
```

This illustrates several features of Python's matrix handling, some nice, some annoying. We created a rank one matrix A by dropping a range into a matrix (remember the range would have been an array), and then premultiplying it by its transpose. Then we set an entire 2×2 block of that matrix to 1 in a single line. We already knew how to do that one element at a time, but by specifying a collection of indices using the notation [1:3,1:3], we were able to refer to the four elements with indices [1,1], [1,2], [2,1], and [2,2] all at the same time, and set them equal to 1.

When we specify a range using the colon notation, there are a few conventions that apply. These are not entirely intuitive.

s:n refers to a range of indices starting with s and ending with the greatest integer less than or equal to n-1.

s: refers to a range of indices starting with s that extends to the last possible index.

:n refers to a range of indices starting with the first possible index, and extending to the greatest integer less than n-1.

: refers to a range of all possible indices in that position.

s:n:i refers to range of indices starting with s and counting in increments of i to the greatest number s+ki that is less than or equal to n-1, where k is a nonnegative integer.

These rules can be mixed and matched in a somewhat rational way. Here are some examples of these rules.

```
>>> A=zeros((5,5))
>>> A
array([[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.]])
>>> A[0,0:5] = arange(5)
>>> A
array([[ 0., 1., 2., 3., 4.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.]])
>>> A[1,:] = -3
>>> A
array([[ 0., 1., 2., 3., 4.],
[-3., -3., -3., -3., -3.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.],
[ 0., 0., 0., 0., 0.]])
>>> A[2:,4] = pi
>>> A
array([[ 0. , 1. , 2. , 3. , 4. ],
[-3. , -3. , -3. , -3. , -3. ],
[ 0. , 0. , 0. , 0. , 3.14159265],
[ 0. , 0. , 0. , 0. , 3.14159265],
[ 0. , 0. , 0. , 0. , 3.14159265]])
>>> A[2,:4]=pi/2
>>> A
array([[ 0. , 1. , 2. , 3. , 4. ],
[-3. , -3. , -3. , -3. , -3. ],
[ 1.57079633, 1.57079633, 1.57079633, 1.57079633, 3.14159265],
[ 0. , 0. , 0. , 0. , 3.14159265],
[ 0. , 0. , 0. , 0. , 3.14159265]])
>>> A[3,1:4]=1.0/3
>>> A
array([[ 0. , 1. , 2. , 3. , 4. ],
[-3. , -3. , -3. , -3. , -3. ],
[ 1.57079633, 1.57079633, 1.57079633, 1.57079633, 3.14159265],
[ 0. , 0.33333333, 0.33333333, 0.33333333, 3.14159265],
[ 0. , 0. , 0. , 0. , 3.14159265]])
>>> A[3:5,:2] = -9
>>> A
array([[ 0. , 1. , 2. , 3. , 4. ],
[-3. , -3. , -3. , -3. , -3. ],
[ 1.57079633, 1.57079633, 1.57079633, 1.57079633, 3.14159265],
[-9. , -9. , 0.33333333, 0.33333333, 3.14159265],
[-9. , -9. , 0. , 0. , 3.14159265]])
```

We started by using the Numpy function we have used before called `zeros` to make a matrix of floating point zeros. This is particularly useful in view of what we have seen regarding integer arithmetic – it is a way to be sure that this array is interpreted by Python as a floating point object.

In each of the assignments we put a number or collection of numbers into some submatrix of `A`. In the first case we put a one-dimensional range object into the first (index 0) row. The range object we made was actually a collection of integers, but because `A` is already a floating point array, it is converted to floating point when it goes in. In the second example, we put a constant into every entry in the second (index 1) row. Again, the integer -3 is converted to floating point when it goes into the array. After that, we put an approximation for π into the last three entries in the last column. Next, $\pi/2$ in the first four entries in the fourth (index 3) row. Another command put an approximation to $1/3$ into the middle three entries of the penultimate row. Notice here that we had to make sure to use a floating point representation for either the 1 or the 3 - otherwise Python would have supposed that it was doing integer arithmetic, and would have rounded the result to zero. The fact that the array was floating point would not have saved us – the arithmetic happens first, the assignment that would convert to floating point only happens *after* that. Finally, we put the number -9 in the small 2×2 block in the lower left corner.

When you understand all these operations, you will be able to manipulate matrices and arrays with speed and subtlety.

7.6 Functions

The real value of Python does not lie in its usefulness as a command-line calculator. Instead, it is because it is a full-featured programming language. The point of this is that you and others can write code blocks that can be reused – write once, use forever. We have already created one function from the command line in Section 7.2.4. However, this approach to writing functions still forces us to type the function anew in each Python session. Instead, we will usually want to type our more complicated functions into separate files which we can then run from the Python command line, or even by themselves.

Consider the following exceptionally simple function. We write it using multiple lines just to remind you that indentations are important.

```
def f(x):
  y = 4+x*(-3.0+x*(1.7+x))
  return y
```

We can type this function into a file using our favorite text editor. We'll call the file `myfunctions.py`. Observe that the .py extension is important - Python assumes that files to be imported have that extension, so if you use something else or have no extension, it might cause problems. Python files end in ".py".

Once we have the function saved in a file, we can import it into Python in the same way we imported Numpy. Start Python from the directory where that file is located, and type the following.

```
>>> from myfunctions import *
>>> f(4)
16
```

As with Numpy itself, we could also import the file as its own object, and then use the name of that object in referring to the functions it contains. There are a few advantages to that in this case. For one thing, you no longer have to worry about replacing an existing Python function with your own – now your function name has e.g. “mf.” prepended. Moreover, chances are that you are working on your functions and debugging them, so you are probably making changes to your file of functions constantly. In that case, you always want to run the most recent version in Python. Python does import the class anew if you use the import statement, but you can use the reload command to get your revised functions.

```
>>> import myfunctions as mf
>>> mf.f(4)
16
>>> reload(mf)
<module 'myfunctions' from 'myfunctions.py'>
```

If you have used Matlab, you know that one annoying thing about it is that functions must always be grouped one to a file, with the filename the same that of the function it contains. That is not true in Python. We can put as many individual functions as we want in the file myfunctions.py. Note, however, that if we want to use Numpy functions in the functions we write ourselves we must import Numpy in the file containing our functions. Here is our new `myfunctions.py` file.

```
from numpy import *
def f(x):
  y = 4+x*(-3.0+x*(1.7+x))
  return y
def g(x):
  return sin(pi*x)+4.0*x*x
```

Now we can reload our file in Python, and then use the new function.

```
>>> reload(mf)
<module 'myfunctions' from 'myfunctions.py'>
>>> mf.g(1.5)
8.0
```

We note in passing that Python is an object oriented language. We can create classes and instance them, and do all the sophisticated things that modern programmers expect. However, this is beyond the scope of the current discussion – we will touch on it briefly later, but not with any intention to learn object oriented programming. Our goal here is much more limited.

There is another possibility in Python. We can actually run our Python code without ever starting the Python interpreter. In this case our Python function essentially becomes a command. We can run the program from the command line by typing `python myfunctions.py`. When we run Python with a file name after it, then it assumes that we are to run the commands in the file and then quit.

If we are just going to run the file, we need some top-level statements to execute. Otherwise the file only defines a couple of functions without using them. We change the file as follows.

```
from numpy import *
def f(x):
  y = 4+x*(-3.0+x*(1.7+x))
  return y
def g(x):
  return sin(pi*x)+4.0*x*x

y=f(g(3.0))
print "f(g(3.0)) = "+str(y)
```

The `print` statement makes sure that the function prints out an answer. Otherwise it would just compute `f(g(3.0))`, assign the value to a variable `y`, and quit. This way we get to see the answer. Thus, we defined two functions `f` and `g`, then told Python to compute `f(g(3.0))` and assign that number to a variable called `y`, and finally to print out a message saying that `"f(g(3.0)) = "` and append to that a string representing the value of `y`. When we run this, we see

```
python myfunctions.py
f(g(3.0)) = 1296.0
```

We should emphasize that the `print` command needs to handle character strings. We could not just append the floating point number `y` to the string – we needed first to convert the number to a character string using the `str` command.

7.7 Flow Control

If we are to use Python to write powerful mathematical programs, we need those programs to be able to make decisions and perform repetitive operations. All programming languages can do those things. The commands that do that are called flow control statements. In particular, every programming language possesses at least two statements that allow us to control the flow of execution. Decisions are made using ”if” statements, while repetitive operations can be done using ”for” statements.

7.7.1 If

The “if” statement in Python has the following structure.

```
if condition1:
  statements to execute
elif condition2:
  more statements
else:
  default statements
```

Remember that in Python, indentations are important! Only the first two lines are really required. The other lines can be added if more options are needed. The execution would go like this: Python looks at `condition1` (e.g. `x<0`) and if it is true, it performs the `statements to execute`. If `condition1` is false, then Python checks `condition2` (e.g. `x>-1`), and if that is true, it executes the `more statements`. If `condition1` and `condition2` are both false, Python executes default statements. In this construction, “if”, “elif”, and “else” are keywords, while the rest is content you will create yourself. Note that “elif” stands for “else if”, i.e., if the previous condition was not true, try this. Let’s look at an example. We want to write a function that will perform the composite trapezoidal rule to approximate the integral of a function designated here by `fct`, i.e. it approximates $\int_a^b \text{fct}(x)\,dx$.

```
def trapezoid(fct,a,b,n):
  if n<=0:
    print "Error: n must be positive"
    return False
```

We first define the function, called `trapezoid`. It takes four arguments: `fct`, `a`, `b`, and `n`. The last number n denotes the number of subintervals used for the composite trapezoidal rule. It seems important that the number n be positive. Otherwise the scheme will fail. Thus, we test n to see if it is less than or equal to zero. If so, then the program does the indented code, which prints an error message and quits the function. When we run this code we get the following results.

```
>>> import pythonfcts as mf
>>> def f(x):
...   return x*x
...
>>> mf.trapezoid(f,0,1,-4)
```

Table 7.2: Comparison Operators

Operator	Meaning
==	...is equal to
!=	...is not equal to
<	...is less than
>	...is greater than
<=	...is less than or equal to
>=	...is greater than or equal to

```
Error: n must be positive
False
```

Python understands several basic comparison operators, as detailed in Table 7.2. It can also combine these using logical "and" and "or" operators. For example, we could have written the "if" statement above equivalently as `if n<or n==0`.

7.7.2 For

The "for" keyword allows us to perform some set of commands repetitively, with a counting variable that changes in some prescribed way. The basic syntax looks like this.

```
for i in list:
  some commands that might depend on i
```

In this, the word "for" must appear exactly as shown. The variable that follows is denoted `i` here, but could have any name. The keyword "in" must come next, and then some list (frequently created as a range or arange) from which the values of `i` appears. The colon at the end of the line is not optional. After that we use some uniform indentation to
put as many commands as we want to be repeated. Here is an example.

```
>>> for i in [-1,1.5,7]:
...   j = i**2
...   print j
...
1 2.25 49
```

In these commands, our counting variable is called `i`, and we tell it to take on the values -1, 1.5, and 7, in turn. Each time it takes one of those values, we compute a variable `j` as the square of `i`, and then print `j`. We could do the list differently, using an arange, viz.

```
>>> from numpy import *
>>> for i in arange(-1,3):
...   j = i**2
...   print j
...
1 0 1 4
```

This time we need Numpy. The point is that “for” is a plain Python construct – we only needed Numpy when we wanted to use an arange.

In our trapezoidal rule example, recall that the trapezoidal rule can be written as

$$\int_a^b f(x)\,dx \approx \frac{h}{2}\left[f(a)+f(b)+2\sum_{i=1}^{n-1} f(x_i)\right] \tag{7.1}$$

$$= \frac{h}{2}\left[-f(a)-f(b)+2\sum_{i=0}^{n} f(x_i)\right], \tag{7.2}$$

when $a = x_0$ and $b = x_n$. We can type this into our function file to make our trapezoid function look like the following.

```
def trapezoid(fct,a,b,n):
  if n<0 or n==0:
    print "Error: n must be positive"
    return False
  h = (double(b)-double(a))/double(n)
  total = 0.0
  for i in arange(n+1):
    total += fct(a+i*h)
  total *= 2.0
  total -= fct(a)+fct(b)
  return total*h/2.0
```

Have we ever mentioned that indentations are very important in Python? The indentations here indicate the code that runs if the condition is true, and the code that runs repeatedly in the “for” construction. After the “if” part we discussed earlier, we initialize a variable called total to zero. This is where we accumulate the sum from the equation. We then use a “for” construction to run through indices `i` starting with 0 and ending with `n`, using these to construct points at which to evaluate the functions. We multiply these function values by two as in the equation, and then subtract off the numbers on the ends.

The next line is not inside the “for” construction, because it is not indented. Thus, it is only executed once. It takes the total and multiplies it by the 2 from the equation. We could have done this inside the “for” construction, but then that would have been n multiplications. This way we have only one multiplication, and we get to illustrate the use of the *= operator. Finally we just subtract `fct(a)` and `fct(b)` from the total. Finally, we multiply the whole total by the quantity outside the brackets in the equation, and send it back to the Python session as the answer.

Now we can run this function in our Python session.

```
>>> reload(mf)
<module 'myfunctions' from 'myfunctions.pyc'>
>>> mf.trapezoid(f,0,1,100)
0.33335000000000004
>>> mf.trapezoid(f,0,1,1000)
0.33333349999999995
```

We used the function `f` that we defined earlier. The correct answer would be $1/3$, so we see that our approximation is correct to four decimal digits with 100 subintervals, and six digits with 1000 subintervals. It seems as if our function works.

7.8 Classes

Python is a so-called object oriented language. We have no intention here of discussing object orientation and classes in detail, but if you can understand just a tiny bit about classes and objects generated from them, then much of the functionality we will see later will make more sense. Thus, please understand this page not as a tutorial in how to create a class, but rather as an illustration of how variables and methods are associated with python objects.

Let's just look at a simple class, and spend some time talking about it. This class abstracts a circle, and provides a couple of methods to evaluate properties of that circle. The following code is all typed into a single file called, say, "circle.py".

```
from numpy import pi,sqrt
class circle:
  def __init__(self,cx,cy,r):
    self.cx = cx
    self.cy = cy
    self.r = r
  def area(self):
    return self.r*self.r*pi
  def inside(self,x,y):
    dist = sqrt((x-self.cx)*(x-self.cx)+(y-self.cy)*(y-self.cy))
    if dist<= self.r:
      return True
    else:
      return False
```

Before we even start properly, we have to import the methods and variables we will need from Numpy. We have to do this even if Numpy was already imported into the Python session where we plan to use our circles. We only imported pi and sqrt, because that was all we needed for this class – we save some memory by not importing everything.

As we know, a circle is completely characterized by its center and radius. The x and y coordinates of the center of the circle, and its radius, are the attributes of this class. Everything else the class does will depend on those attributes. Thus `cx`, `cy`, and `r` are variables associated with the class. Nothing else in Python can see those variable names unless it refers to a specific circle object. Because they are particular attributes of the class, they must be invoked using a reference to the class itself, even from within the class. This is why those attributes are referred to using e.g. the `self.cx` notation. The self variable is always available inside a class as a way to identify itself. Note that we *did* need to list the self as the first argument in every method of the class.

There is a second kind of attribute associated with a class: the method attribute. These are functions that are part of the class. Like variable attributes, method attributes are called by prepending the name of a class object and a period to the name of the function. The circle class has three method attributes: `area`, `inside`, and `__init__`. The last is a very special method. It is invoked any time an object of this class is created. The idea is that every circle requires a center and radius, so we should specify those when we create any circle.

The class itself is an abstraction – it is the idea of a circle. To make an actual object based on the class, we define a Python variable that is identified as a circle.

```
>>> from circle import *
>>> C=circle(1,-1,3)
>>> C.cx
1
>>> C.cy
-1
```

This creates a circle object called `C`. The circle `C` has center (1,-1) and radius 3. These numbers are installed in the attributes `cx`, `cy` as soon as the object `C` is created, through the arguments we passed to the circle class at that time. After the circle is created, we can gain access to its variable attributes using the notation `C.cx` and so on. We note in passing that, unlike other languages, Python provides no way to hide or protect variable attributes from the Python session. If you have not seen other object oriented languages, pay no attention to the last remark.

We can make as many different objects based on our class as we like. Below we create a second object using the circle class. Once we have an object based on a class, we can use the method attributes to perform calculations or acquire information about that object.

```
>>> D=circle(2,0,2)
>>> C.area()
```

```
28.274333882308138
>>> D.area()
12.566370614359172
>>> C.inside(-1,0)
True
>>> D.inside(-1,0)
False
```

We refer to each method attribute of an object using the notation object_name.method_name. This means that in our case, `C.area()` is different from `D.area()`, even though the code for those methods was the same. The methods we called refer to the different variable attributes for their respective objects.

Thus, we see that a class object is a way of encapsulating some variables together with all the functions required to manage and maintain those variables. In this discussion we will not need to write our own classes (unless we want to), but we will see many classes that are part of other packages we will load.

7.9 Strings

Like all modern languages, Python has a powerful ability to store and manipulate character strings. This is generally less important to us as mathematicians, but nonetheless we often have to label axes, title plots, and use strings in other ways, so we'll look at a few of the things we can do.

As discussed in the data type page, Python can recognize strings entered using any of three types of quotation marks. Once they are entered, they can be manipulated in fairly natural ways using operators that we would ordinarily consider as applying only to arithmetic.

```
>>> y="Yossarian said, "
>>> twice='"I see everything twice!"'
>>> y+twice
'Yossarian said, "I see everything twice!"'
>>> twice*2
'"I see everything twice!""I see everything twice!"'
```

We see that we can define two strings as we did on the earlier page, but then concatenate them using a + symbol: in Python, adding two strings concatenates them. Multiplying a string by an integer concatenates that string to itself the specified number of times. Do not try to multiply one string by another.

Every string is actually an object in Python, with its own associated methods that can return properties of the string, or change the string in various ways.

```
>>> len(y)
16
>>> y.find('said')
10
```

```
>>> y[10]
's'
>>> Y=y.replace('said','shouted')
>>> Y
'Yossarian shouted, '
>>> Y.upper()
'YOSSARIAN SHOUTED, '
>>> z=Y.lower()
>>> z
'yossarian shouted, '
>>> z.capitalize()
'Yossarian shouted, '
```

One of the methods associated with a string is `format`. This allows us to create strings on the fly, using the values of variables we might have calculated. In this case, we can specify the string using braces {} where we want to make substitutions according to the order of arguments to format. This is easier to see than to describe.

```
>>> call="Call me {}."
>>> call.format("Ishmael")
'Call me Ishmael.'
>>> call.format("maybe")
'Call me maybe.'
```

We can use more than one set of braces if we like, in which case the arguments to format are substituted in order.

```
>>> times="It was the {} of times, it was the {} of times."
>>> times.format("best","worst")
'It was the best of times, it was the worst of times.'
>>> times.format("worst","best")
'It was the worst of times, it was the best of times.'
```

7.10 Speed

In our discussion of flow control we wrote a Python function to compute a trapezoidal rule approximation to the integral function we specified over some interval we also specified. That approximation was given by

$$\begin{aligned}\int_a^b f(x)\,dx &\approx \frac{h}{2}\left[f(a)+f(b)+2\sum_{i=1}^{n-1} f(x_i)\right] && (7.3)\\ &= \frac{h}{2}\left[-f(a)-f(b)+2\sum_{i=0}^{n} f(x_i)\right], && (7.4)\end{aligned}$$

where $a = x_0$ and $b = x_n$. The function we used to evaluate this approximation was

```
def trapezoid(fct,a,b,n):
  if n<0 or n==0:
    print "Error: n must be positive"
    return False
  h = (double(b)-double(a))/double(n)
  total = 0.0
  for i in arange(n+1):
    total += fct(a+i*h)
  total *= 2.0
  total -= fct(a)+fct(b)
  return total*h/2.0
```

It seems interesting to find out how fast this function is. We can do that using the `clock` method from the time module. This function gives the current time to some level of precision. On our computer it gave the time to the nearest hundredth of a second. We should be able to find the time before we start our trapezoid function, then run the trapezoid function, then find the time again. The difference between the two times is roughly how long our function took to run. We can write a little function to do this for us.

```
>>> import myfunctions as mf
>>> import time
>>> def ttrap(trap_fct):
...   t1=time.clock()
...   trap_fct(mf.f,0,1,1000000)
...   t2=time.clock()-t1
...   print t2
...
>>> ttrap(mf.trapezoid)
11.58
```

Our function is called `ttrap`, and it takes one argument: the name of the function it is going to time. It finds the current time and puts it in a variable called `t1`. Then it runs the function you supplied as an argument. After that it finds the time again, subtracts `t1` to find the elapsed time, and prints that. We find that running our trapezoid function with a million subintervals (`n`=1000000) takes eleven and a half seconds.

You might wonder why we were so cagey about the name of the function we wanted to time – why not just type trapezoid instead of `trap_fct` and then supplying `trapezoid` as an argument. You might also wonder whether we could write a faster function. That, of course, is the point. We can write a function that runs faster, and we'll give it a different name so we can use this `ttrap` function to time it.

The thing that takes all the time in the trapezoid function is the "for" construction. A million times it has to evaluate $f(x)$ for some scalar x. We could write this instead to do the calculations on a Numpy array. Maybe that would go faster. Our new function looks like this.

```
def arraytrapezoid(fct,a,b,n):
  if n<0 or n==0:
    print "Error: n must be positive"
    return False
  h = (double(b)-double(a))/double(n)
  x = arange(a,b+h,h)
  y = 2.0*fct(x)
  y[0] = y[0]/2.0
  y[n] = y[n]/2.0
  return sum(y)*h/2.0
```

Everything is the same up through the definition of h. At that point, instead of starting a “for” construction, we define a vector x of points of the partition, in such a way that $x_i = a + ih$ for each $i = 0, 1, ..., n$. After that, we have only to evaluate the function for each point of that array, which we do simply by feeding the entire array to fct as an argument. The properties of arrays take care of the elementwise multiplication that is required. After that we just subtract the extra function value at a and b, and return the result.

Evidently this version of the code needs a great deal more memory, since it has to store the entire vector x as well as the function evaluation over that: y. However, this version of the code is spectacularly faster than our earlier version. Remember that our other version took over eleven seconds to run.

```
>>> ttrap(mf.arraytrapezoid)
0.05
```

Why is this more than twenty times faster? It is the difference between interpreted code and compiled code. These are essentially the two ways to run computer programs. Interpreters look at the text commands you type, parse their meaning, translate them into machine code, and then execute that machine code. Compilers translate an entire file of code into machine language, which then only needs to be loaded and run full speed. Interpreted code thus runs much more slowly than that compiled, “pretranslated” code.

In our case, the `trapezoid` function relies on that “for” construction. Thus each time it executes the code, the Python interpreter must translate that command to add to the total, then execute it. By contrast, in the `arraytrapezoid` function, everything happens through the Numpy array construction. This, like many things in Numpy, is compiled. Thus in this case, instead of one million interpreted lines, there is only one, and then the computation happens in compiled code, so it is much faster. In general, because the Numpy package contains much compiled code, the more we rely on it and not native Python, the faster our programs will run.

7.11 Plots

The title of this section is far too grandiose: we will discuss only some simple aspects of the Pylab package for Python. Pylab is a collection of functions designed to mimic much of the function of Matlab in Python, and in particular, to mimic Matlab's powerful plotting capabilities. Pylab imports and capitalizes on Numpy and another Python package called Matplotlib. Indeed, you will probably find henceforth that you simply load Pylab at
the beginning of every session and leave it at that.

In order to illustrate the use of the plotting facilites in Pylab, let's write a Python function to make a classic picture. We'll plot a function passed as an argument from a Pylab session, over an interval specified as arguments to the Python function, and then plot the rectangles of a Riemann left sum to approximate the integral of the given curve. We start with a function to plot some curve. In our file "myfunctions.py" we change the import line at the top to `from pylab import *`. This replaces the line we had before that imported Numpy. We type the following code e.g. at the end of the file.

```
def riemannplot(fct,a,b,n):
  if n<=0:
    print "Error: n must be positive"
    return False
  smoothh= (b-a)/100.0
  x = arange(a,b+smoothh,smoothh)
  plot(x,fct(x))
  show()
```

We recycled code to make sure that n is positive but then computed a reasonably fine array of points where we want to plot the function given as fct. Then we just call the plot function. The plot command requires two arguments: an array of abscissæ (x-coordinates) for the points to be plotted, and an array of ordinates (y-coordinates). The `plot` command creates the figure, but does not display it on the screen. To do that you must call the `show` command. This allows us to put more curves on the plot being created, or alter it in other ways.

We call our new function as follows.

```
>>> import myfunctions as mf
>>> def f(x):
...   return x**3-2*x**2+0.5*x+0.5
...
>>> mf.riemannplot(f,-1,2,8)
```

At this point a new window appears on our screen, containing the plot we made. The plot is shown in Figure 7.1. Figure 1

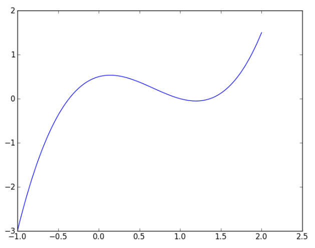


Figure 7.1: A simple plot in Pylab

Next, we want to put in all the boxes showing the elements of the Riemann sum. We can use the `bar` command for this. The `bar` command again takes two arguments. The first is an array of x-coordinates for the left corner of each bar, and the second is a height of the bar, which can be negative. Thus, we make a new variable h that represents the width of each bar, and then use that to create an array `riemannx` of x-coordinates for the left corners of the bars. The array riemanny is just the value of the input function fct at those points. Thus, the picture we are trying to draw corresponds to a Riemann left sum. The picture we actually drew is ugly – it appears in Figure 7.2. Our code appears below.

```
def riemannplot(fct,a,b,n):
  if n<=0:
    print "n must be positive"
    return False
  smoothh= (b-a)/100.0
  x = arange(a,b+smoothh,smoothh)
  plot(x,fct(x))
  h = (double(b)-double(a))/double(n)
  riemannx = arange(a,b,h)
  riemanny = fct(riemannx)
  bar(riemannx,riemanny)
  show()
```

The problems with this new picture include our inability to see the original curve through the bars; the bars are too wide; and we would really like the bars to be a different color to contrast with the curve. We can fix all these things by adding attributes to our bar command. This means that we add arguments to the command, each of which has the form `name=value`. In our case, the three things we want to change include the color: `facecolor='orange'`. The width of the bars should be h, so we can add an argument `width=h`. Finally, we can make the bars 50% transparent by setting `alpha=0.5`. Our bar command now looks like

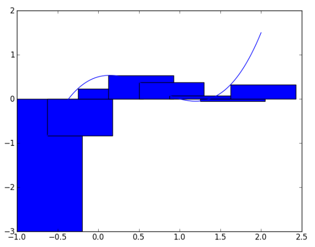


Figure 7.2: A combined plot in Pylab

```
bar(riemannx,riemanny,width=h,alpha=0.5,facecolor='orange')
```

The new plot appears in Figure 3.

We should probably also label the axes, maybe provide a title, and maybe restrict the window to the part we care about. In particular, the default plot ranged out to 2.5 for reasons we do not fully understand. We'll restrict it again. Here is the code.

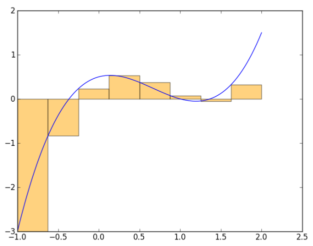
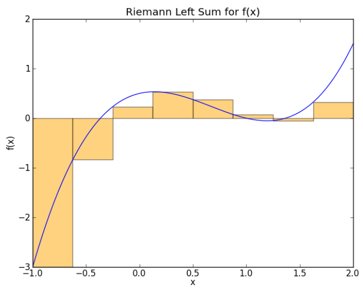


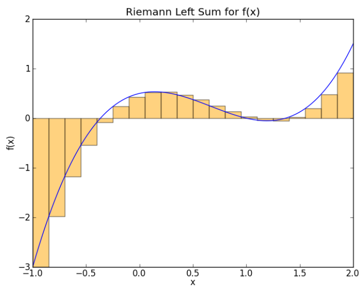
Figure 7.3: Appearance modifications

```
def riemannplot(fct,a,b,n):
  if n<=0:
    print "n must be positive"
    return False
  smoothh= (b-a)/100.0
  x = arange(a,b+smoothh,smoothh)
  plot(x,fct(x))
  h = (double(b)-double(a))/double(n)
  riemannx = arange(a,b,h)
  riemanny = fct(riemannx)
  bar(riemannx,riemanny,width=h,alpha=0.5,facecolor='orange')
  xlabel('x') ylabel('f(x)') title('Riemann Left Sum for f(x)')
  xlim(-1,2) show()
```

The commands we just added are pretty intuitive. Note in particular the xlim command, which takes two arguments: the left limit; and the right limit. There is a similar ylim command that we could have used, had we wanted to. The picture appears in Figure 7.4a.



(a) A completed plot



(b) A plot with more subintervals

At this point we have a function that has some wide applicability. For example, we can use it to see how the Riemann sum improves as we refined the partition by increasing the number of points. In Figure 7.4b we see the results of a call to the function with `n=20`.

```
>>> mf.riemannplot(f,-1,2,20)
```

7.12 Symbolics

In the twenty-first century there is little or no reason to compute complicated integrals and derivatives by hand. We can instead use any of various Python packages to do exact arithmetic, perform algebraic operations, and evaluate limits, integrals and other calculus constructions easily. One package for Python that is somewhat mature is called Sage. This depends upon Python, but builds on top of it, rather than being a simple package such as Numpy. We will instead use Sympy, a relatively new and less than mature package with a bright future. Ultimately, the point is that symbolic math packages are a dime a dozen these days, and learning one will demonstrate most of the ideas anchoring all of them, though the details of syntax vary from one package to another.

The first thing to note about Sympy is that it prefers to treat numbers as exact. This means that each number is an object in itself, with various means of interpretation, including approximation to an arbitrary number of digits of accuracy. Thus, Numpy provides a fixed floating point approximation to π, but Sympy treats π as a real number that can be evaluated in floating point arithmetic to any number of digits you require.

```
>>> from numpy import *
>>> pi 3.141592653589793
>>> from sympy import *
>>> pi
pi
>>> pi.evalf()
3.14159265358979
>>> pi.evalf(50)
3.1415926535897932384626433832795028841971693993751
```

There are several significant things here. First, when you do arithmetic with sympy objects you might not get the simple numbers you expect – learn to change your expectations. The second thing is that working with symbolic objects will be much slower than working with ordinary Python numbers. Do not use Sympy objects unless you really need to do exact or symbolic calculations.

Another thing to be careful with here is that if it is not clear that the object we are dealing with is a Sympy object, then ordinary python rules apply. Thus, 1/3 is treated as an integer arithmetic operation, with result 0. Likewise, 1.0/3 is treated as a floating point operation, with result 0.3333333333333333. If we want to do an exact calculation, we must make a rational expression. As soon as Python realizes it is working with one rational, it treats an entire expression as if it is rational.

```
>>> q=Rational(1,3)
>>> q
1/3
>>> q.evalf(50)
```

```
0.3333333333333333333333333333333333333333333333333
>>> q=Rational(1)/3
>>> q
1/3
```

The whole point of working with a symbolic package is to permit us to do calculations with symbolic variables. Sympy allows us to define symbolic variables and then work with them. There are two ways to start this: we can use the symbols command, or we can use the var command. New variables formed using symbolic variables are themselves symbolic.

```
>>> x,y=symbols('x,y')
>>> var('u,v')
(u, v)
>>> x-x
0
>>> x+x
2*x
>>> expr=(u+v)**4
>>> expr (u + v)**4
```

Symbolic variables have many functions associated with them that allow us to do various algebraic and calculus operations. The whole point of using e.g. the var command to declare the variable was for python to create an entire class around that name, together with associated mathematical associations and operations. We get to these using the usual method notation: the variable name, followed by a period and the name of the method we want to apply. For example, the expand function expands polynomial expressions. We can reverse this by using factor. If we need to substitute one expression for another, there is a subs method to do that.

```
>>> newexpr=expr.expand()
>>> newexpr
u**4 + 4*u**3*v + 6*u**2*v**2 + 4*u*v**3 + v**4
>>> newexpr.factor()
(u + v)**4
>>> newexpr.subs(u,2)
v**4 + 8*v**3 + 24*v**2 + 32*v + 16
>>> (newexpr.subs(u,2)).factor()
(v + 2)**4
```

Notice in the last expression that we simply put parentheses around a computed quantity, which was symbolic because it was created from symbolic expressions, and then used a method associated with this new symbolic object. Once again for emphasis: any time we perform a computation that uses a symbolic variable, the result is itself symbolic, and hence has access to all the methods associated with symbolic variables.

When we use symbolic expressions, we really don't want to have to look at all that "**" stuff. We want to see something that resembles ordinary mathematical notation. Many symbolic packages call this "pretty printing". Regrettably, the result is not always pretty, but most of the time it produce result that are more legible than the straight python syntax. To do pretty printing using Sympy, run three commands in your session:

```
>>> import sys
>>> oh=sys.displayhook
>>> sys.displayhook=pprint
```

After that is done, your expressions should be "pretty printed".

```
>>> u=1/((y+3)*(y-4)**2)
>>> u 1/((y - 4)**2*(y + 3))
>>> import sys
>>> oh=sys.displayhook
>>> sys.displayhook=pprint
>>> u
      1
----------------
      2
(y - 4) *(y + 3)
```

In all further discussion of the Sympy package, we will suppose that you have run these commands.

Sympy can manipulate fractional expressions. For example, it contains a command that every symbolic manipulation program in history has had: `simplify`.

```
>>> simplify(newexpr/(u+v))
 3     2          2   3
u + 3*u *v + 3*u*v + v
>>> (newexpr/(u+v)).simplify()
 3     2          2   3
u + 3*u *v + 3*u*v + v
>>> trigexp=sin(u)**2+cos(u)**2
>>> trigexp
   2        2
sin (u) + cos (u)
>>> trigexp.simplify()
1
```

Simplify looks for common factors in the numerator and denominator of an expression and cancels them if they are found. It also tries to apply some simple rules about exponents and trigonometric functions.

Another requirement in dealing with fractional expressions is combining and breaking up objects with different denominators. For this Sympy gives us `together` and `apart`. The point is that `apart` is doing partial fraction decompositions for you.

```
>>> expr = 1/((u+1)*(u-3))
>>> newexpr=apart(expr)
>>> newexpr
      1           1
- --------- + ---------
  4*(u + 1)   4*(u - 3)
>>> together(newexpr)
       1
---------------
(u - 3)*(u + 1)
```

Bibliography

[1] *Mathworks - matlab and simulink for technical computing*, October 2013. `http://www.mathworks.com`.

[2] *Wolfram—alpha: Computational knowledge engine*, October 2013. `https://www.wolframalpha.com/`.

[3] *National marine fisheries service*, June 2016. `http://www.nmfs.noaa.gov/`.

[4] *Siam information for authors*, June 2016. `https://www.siam.org/journals/auth-info.php`.

[5] W. C. DAVID KINKAID, *Numerical Analysis*, Brooks Cole, Pacific Grove, California, 1991.

[6] L. KLEINROCK, *Information Flow in Large Communication Nets*, ph.d. thesis proposal, Massachusetts Institute of Technology, July 1961.

[7] D. E. KNUTH, *The TEX book*, Addison-Wesley, Reading, 1984.

[8] L. LAMPORT, *LaTeX: A Document Preparation System*, Addison-Wesley, Reading, 2nd edition ed., 1994.

[9] M. D. SPIVAK, *The Joy of TeX, a Gourmet Guide to Typesetting with the AMSTeX Macro Package*, AMS, Providence, 2nd edition ed., 1990.

[10] R. E. K. VINTON G. CERF, *A protocol for packet network intercommunication*, IEEE Transactions on Communications, 22 (1974), pp. 637–648.

[11] J. H. WILKINSON, *The Algebraic Eigenvalue Problem*, Oxford University, Oxford, 1965.